AD-A270 137

4

Final Report

Volume 3

A Parallel and Pipelined Architecture for Estimation of Direction of Arrival using a Bilinear Transformation Method.

Submitted to:

Grant No. N00014-91-J-1011
Department of the Navy
Office of the Chief of the Naval Research
Arlington, VA 22217-5000

Submitted by:

M. M. Jamali Principal Investigator
S. C. Kwatra Co-Investigator
Ravindranath Suria Research Assistant

Department of Electrical Engineering
College of Engineering
The University of Toledo
Toledo, Ohio 43606

Report No. DSPH-3
August 1993

Final Report

Volume 3

A Parallel and Pipelined Architecture for Estimation of Direction of Arrival using a Bilinear Transformation Method.

Submitted to:

Grant No. N00014-91-J-1011
Department of the Navy
Office of the Chief of the Naval Research
Arlington, VA 22217-5000

Submitted by:

M. M. Jamali Principal Investigator
S. C. Kwatra Co-Investigator
Ravindranath Suria Research Assistant

Department of Electrical Engineering
College of Engineering
The University of Toledo
Toledo, Ohio 43606

Report No. DSPH-3
August 1993

This report contains part of the work performed under ONR grant N00014-91-J-1011 during the period October 1990 to July 1993. The research was performed as part of the Masters thesis requirement of Mr. Ravindranath Suria.

M. M. Jamali

Principal Investigator

Abstract

High resolution direction-of-arrival (DOA) estimation is important in many sensor systems. It is based on the processing of the received signal and extracting the desired parameters of the DOA of plane waves. The estimation of angle of arrivals of multiple broadband sources has been carried out in a variety of ways over the past few years. In this research an algorithm for broadband DOA estimation using a simple bilinear transformation matrix is investigated and a parallel and pipelined architecture is developed. When compared to other coherent approaches, this algorithm has the advantages of being non-iterative and does not require any initial estimates of the angles of arrival and all angles are computed from a single step of coherent subspace calculations. Hence it is a very suitable algorithm for computation of DOA using dedicated hardware. The advances in the area of Very Large Scale Integration (VLSI) have made it possible to design special purpose hardware which has the advantage of very high speed and overall lower system cost when compared to a system which runs off a general purpose computer.

The algorithm is first analyzed and modified to exploit the maximum parallelism in the computations. Each part is simplified and made suitable for execution with special purpose hardware. The design considered the tradeoffs between the timing requirements and the number of processors in each stage. The final part of the research is the design and implementation of a generalized covariance matrix processor for several DOA algorithms, namely the bilinear transformation method, the BASS-ALE method and the narrowband MUSIC algorithm. A VHDL simulation of the processor was done with PowerView, the Sun workstation based CAD tool from ViewLogic. The processor was simulated and layed out using GDT the IC design package from Mentor Graphics.

CONTENTS

List of Figures

CHAPTER 1

Introduction

1.1 ARRAY SIGNAL PROCESSING

The estimation of the direction of arrival (DOA) in sensor systems has been one of the frequently considered problems in digital signal processing. The algorithms used to compute the DOA are based on the processing of the received signal and extracting the desired parameters to estimate the direction of arrival.

Traditionally the approaches to this problem have been separated into the narrowband case which assumes that the signals can be considered to have only one frequency component and the broadband or wideband case in which the signal is considered to consist of a band of frequency components. So far the narrowband case has engendered the maximum interest and a lot of algorithms have been used to achieve the results.

Most narrowband approaches use the so called maximum likelihood (ML) and the maximum entropy (ME) methods [1-3]. The most popular methods for narrowband estimation are the Multiple Signal Classification (MUSIC) and the Estimation of Signal Parameters by Rotational Invariance Techniques (ESPRIT) algorithms [4,5]. Computationally they are the most efficient and hence are considered the most promising candidates to perform the required functions.

The estimation of angle of arrivals of multiple broadband sources has been carried out in a variety of ways over the past few years. The conventional approach is to form a generalized correlator [6] to estimate the Time Difference Of Arrival (TDOA) of the signal at the sensors. Some methods are similar to the narrowband case. The so called maximum likelihood based methods [7-9] require knowledge of the source and noise spectra and are computationally expensive. The parameter estimation based methods [10-12] , assume Auto-Regressive Moving Average (ARMA) models for the received signals and the estimated ARMA parameters are utilized for the TDOA calculations. Such model based methods have computational complexity and their effectiveness depends upon the accuracy of the model chosen to represent the unknown broadband signals. Another way is use a eigendecomposition approach for the estimation. This approach involves the incoherent combination of the eigenvectors of the estimated spectral density matrices at each frequency bin to calculate the TDOAs. One way [13,14] is to use the initial estimates of the angles of arrival to transform the eigenspaces at different frequency bins and generate a single coherent subspace which is eigendecomposed to give more accurate estimates. Well separated angles can be estimated by focusing at different angles at each time and iterating to obtain the accurate results. Most of these methods use algorithms that principally operate in the time domain and have the disadvantage of either needing initial estimates of the angles of arrival or having to perform several iterations before arriving at the result.

1.2 A BROADBAND DOA ALGORITHM

Shaw and Kumaresan [15], proposed an algorithm for broadband DOA estimation using a simple bilinear transformation matrix. An approximation resulting from a dense and equally spaced array structure is used to combine the individual narrowband frequency matrices for coherent processing.

When compared to other coherent approaches, this algorithm has the advantages of being non-iterative and does not require any initial estimates of the angles of arrival and all angles are computed from a single step of coherent subspace calculations. Hence it was found to be a suitable algorithm for computation of DOA using dedicated hardware.

The first objective of this research is to modify and parallelize this algorithm so that maximum computational effectiveness can be exploited. The algorithm is broken up into computational units and various architectures from systolic arrays to MIMD machines are considered for each module. The most appropriate one is presented and a complete system is developed for the whole algorithm.

The next part of the thesis deals with the implementation of a combined covariance matrix processor. The computation of the covariance matrix is a common step in all DOA algorithms. Along with the bilinear transformation algorithm, the narrowband MUSIC [16] algorithm and the broadband BASS-ALE method [17] are considered and and a processor is developed which is capable of computing the covariance matrix for any of the three algorithms. The processor is simulated at the architectural level using VHDL.

The VLSI implementation of the processor is considered next and an ASIC chip is proposed which would contain covariance matrix processor. The detailed design of the processor is performed and the processor is simulated at the gate level using GDT[21-24].

Chapter 2 explains the adapted version of the broadband bilinear transformation algorithm proposed by Shaw and Kumaresan. The authors present a generalized algorithm which has been adapted for using in a system with eight sensors. Chapter 3 explains the modifications to the algorithm which will make it more computationally efficient. This includes the introduction of the Cholesky Decomposition, the modularization and the parallelization of the algorithm so that it can be easily implemented with a parallel architecture. Chapter 4 deals with the design of the system architecture. The proposed architectures for the various modules are shown and the processing elements at each stage are described. Chapter 5 describes the design of a combined covariance matrix processor which has been explained above. The behavioral simulation of the processor is also described and the results are shown. Chapter 6 deals with the VLSI implementation of the processor and the gate leevel simulations that were done. The results and conclusions are presented in Chapter 7. The appendices contain the Fortran code for the algorithm simulation and the VHDL code for the behavioral simulation of the various modules.

CHAPTER 2

The Bilinear Transformation Algorithm

2.1 INTRODUCTION

The thesis is broadly based upon a novel DOA estimation approach proposed by Shaw and Kumaresan[15]. This algorithm estimates the DOA of broadband sensor signals by using a simple bilinear transformation matrix. In this algorithm approximation resulting from a dense and equally spaced array structure is used to combine the individual narrowband frequency matrices for coherent processing. This algorithm is non-iterative and does not require any initial estimates of the angles of arrival. This algorithm has been adapted for use in an eight sensor system. The basic concept of the algorithm and the mathematical transformations it involves is presented in this chapter.

The system that is considered consists of a linear array of 8 sensors which are spaced at equal intervals. They therefore receive signals that are slighty different from each other. The spatial difference in the position of the sensors is reflected by a proportional phase shift in the observed signals at the different sensors. The noise at the sensors is also considered and an additive component is chosen to represent the effects of all small sources. They can be combined and modelled as a Gaussian and stationary process by using the central limit theorem.

2.2 PROBLEM FORMULATION

Consider a linear array with 8 sensors which are spaced at equal distances. The incoming signal is assumed to be composed of d plane waves emitted from d sources ($d < 8$), with an overlapping bandwidth of B Hz. The signal from the kth sensor is expressed as

$$r_k(t) = \sum_{i=1}^{d} s_i(t - (k-1)\frac{\Delta}{c}\sin\theta_i) + n_k(t) \tag{2.1}$$

$$-\frac{T}{2} \le t \le \frac{T}{2} \qquad 1 \le k \le 8$$

where $s_i(.)$ is the signal radiated by the ith source, Δ is the separation between the sensors, c is the propagation velocity of the signal wavefront, θ_i is the angle that the ith wavefront makes with the line of array and n_k is the additive noise at the ith sensor.

Performing the FFT and representing the two sides by their Fourier coeffficents

$$R_k(w_l) = \sum_{i=1}^{d} e^{-jw_l(k-1)\frac{\Delta}{c}\sin\theta_i} S_i(w_l) + N_k(w_l) \tag{2.2}$$

with $w_l = \frac{2\pi}{T}l$, $l = l_1, \ldots, l_1+n_f$, where w_{l_1} and w_{l_1-nf} are the frequencies which span the bandwidth B.

Writing in the matrix notation

$$\mathbf{R}(w_l) = \mathbf{A}(w_l)\,\mathbf{S}(w_l) + \mathbf{N}(w_l) \tag{2.3}$$

where these matrices are composed of the column vectors

$$\mathbf{R}(w_l) = [\mathbf{r}_1(w_l) \ldots \mathbf{r}_8(w_l)]^T \tag{2.4a}$$

$$\mathbf{N}(w_l) = [\mathbf{n}_1(w_l) \ldots \mathbf{n}_8(w_l)]^T \tag{2.4b}$$

$$\mathbf{S}(w_l) = [\mathbf{s}_1(w_l) \ldots \mathbf{s}_d(w_l)]^T \tag{2.4c}$$

and the matrix $\mathbf{A}(w_l)$ is a $8 \times d$ direction finding matrix

$$\mathbf{A}(w_l) = \begin{bmatrix} 1 & \cdots & 1 \\ e^{-jw_l \tau_1} & \cdots & e^{-jw_l \tau_d} \\ \cdots & \cdots & \cdots \\ e^{-j7w_l \tau_1} & \cdots & e^{-j7w_l \tau_d} \end{bmatrix} \tag{2.4d}$$

$$\tau_i = \frac{\Delta}{c} \sin \theta_i \tag{2.4e}$$

τ_i being the TDOA of the ith source. Assuming that the observation time is large enough when compared to the correlation time of the processes, the covariance matrix of the Fourier coefficient vector $\mathbf{r}(w_l)$ will approach the spectral density matrix

$$\mathbf{K}(w_l) = \mathbf{A}(w_l)\mathbf{P}_s(w_l)\mathbf{A}^H(w_l) + \sigma_n^2\, \mathbf{P}_n(w_l) \tag{2.5}$$

where $\mathbf{K}(w_l)$, $\mathbf{P}_s(w_l)$ and $\mathbf{P}_n(w_l)$ are the spectral density matrices of the processes $r_i(.)$ $s_k(.)$, $n_i(.)$ respectively. The noise process is assumed to be independent of the sources and the noise spectral density matrix except for a multiplicative constant σ_n^2.

The problem now reduces to the estimation of the τ_i' s from the covariance matrices $\mathbf{K}(w_l)$ and the noise representations. Then the angles of arrival can be computed from the Equation (2.4e).

2.3 PROBLEM SOLUTION

This particular approach utilizes a bilinear transformation and dense array approximation to form the transformation matrices. The bilinear transformation matrix that is used can be synthesised from the coefficients of the polynomials $p_k(z) = (1+z)^{M-k}(1-z)^{k-1}$, where k =1, 2, ... M-1. M here indicates the number of sensors that the system is using, which in this case is equal to 8. Hence the transformation matrix in this case is an 8x8 matrix, the synthesis of which is shown in the next section.

$\mathbf{E}(w_l)$ denotes a diagonal matrix given by

$$\mathbf{E}(w_l) = \begin{bmatrix} (1+e^{-jw_l r_1})^7 & & & & \\ & \cdots & & & \\ & & \cdots & & \\ & & & \cdots & \\ & & & & (1+e^{-jw_d r_d})^7 \end{bmatrix} \qquad (2.6)$$

Premultiplying $\mathbf{A}(w_l)$ by the transformation matrix $\mathbf{B}$ and simplifying the product gives

$$\mathbf{BA}(w_l) = \begin{bmatrix} 1 & \cdots & 1 \\ j\tan\frac{w_l r_1}{2} & \cdots & j\tan\frac{w_d r_d}{2} \\ \cdots & \cdots & \cdots \\ (j\tan\frac{w_l r_1}{2})^7 & \cdots & (j\tan\frac{w_d r_d}{2})^7 \end{bmatrix} \mathbf{E}(w_l) \qquad (2.7)$$

Assuming that the sensor to sensor separation Δ is small when compared to the wavelengths of the incoming signals, $\tan\frac{w_l r_1}{2}$ can be approximated by $\frac{w_l r_1}{2}$.

Now consider an 8x8 diagonal matrix $\mathbf{D}(\frac{w_c}{w_l})$ whose (k,k) th term is given by

$$d_{kk} = \left(\frac{2w_c}{jw_l}\right)^{k-1} \qquad (2.8)$$

where $w_c = 2\pi f_c$ and f_c is the midband frequency of the signals.

It can be approximated as

$$\mathbf{D}\left(\frac{w_c}{w_l}\right)\mathbf{B}\mathbf{A}(w_l) = \begin{bmatrix} 1 & \dots & 1 \\ w_c r_1 & \dots & w_c r_d \\ \dots & \dots & \dots \\ (w_c r_1)^7 & \dots & (w_c r_d)^7 \end{bmatrix} \mathbf{E}(w_l) \qquad (2.9)$$

There is a new matrix $\mathbf{A}(w_c)$, whose columns are the transformed direction frequency vectors which are dependent upon w_c rather than w_l. The columns of the matrix are linearly independent as long as $r_i \neq r_k$ for $i \neq k$.

A new transformation matrix is defined as

$$\mathbf{T}\left(\frac{w_c}{w_l}\right) = \mathbf{D}\left(\frac{w_c}{w_l}\right)\mathbf{B} \qquad (2.10)$$

This does not depend upon the arrival angles and can hence be computed independently of the angles. Using these transformation matrices for each individual narrowband frequency, all the spectral estimates can now be combined at the midband frequency in the following manner;

$$\mathbf{G} = \sum_{l=l_1}^{l_1+n_f} \mathbf{T}(w_l)\,\mathbf{K}(w_l)\,\mathbf{T}^H(w_l) \qquad (2.11)$$

and $$\mathbf{G_n} = \sum_{l=l_1}^{l_1+n_f} \mathbf{T}(w_l)\,\mathbf{P_n}(w_l)\,\mathbf{T}^H(w_l) \qquad (2.12)$$

Then the coherent signal subspace theorem for the matrix pencil $(\mathbf{G},\mathbf{G}_n)$ is used to estimate all the angles of arrivals by computing the maximas of the measure given by

$$J(\theta) = \frac{1}{\sum_{k=d+1}^{8} \left|\left| \mathbf{a}_\theta(w_c)\,\mathbf{e_k}(w_c) \right|\right|^2} \qquad (2.13)$$

where $\mathbf{e_k}(w_c)$ denotes the generalized eigenvectors of the matrix pencil $(\mathbf{G}, \mathbf{G}_n)$, which correspond to the 8 - d eigenvalues, and $\mathbf{a}_\theta(w_c)$ represents the new direction frequency matrix.

CHAPTER 3

Algorithm Parallelization and Modification

3.1 INTRODUCTION

The first objective in the implementation of signal processing algorithms such as the one outlined in the previous chapter, is to modify them in such a way so that the maximum possible parallelism and pipelining can be achieved which would enable the real time implementation of the algorithm. The modification of the algorithm outlined in the previous chapter takes into consideration the various tradeoffs involved in the ultimate realization of the hardware like the timing and cost considerations which would make the project viable.

Figure 3.1 shows the mathematical transformations that the algorithm involves. The algorithm has been modified into discrete blocks so that the system design can be done in a modular fashion. The sequence of steps involved are as follow:-

1. Collection of sensor samples.
2. Computation of FFT of the samples.
3. Formation and averaging of covariance matrices.
4. Computation of the **G** and $\mathbf{G}_n$ matrices.
5. Perfoming the Cholesky Decomposition.
6. Performing the eigendecomposition.
7. Obtain eigenvalues and eigenvectors
8. Estimate number of sources and angles of arrival.

The main modification introduced in the algorithm involves steps 5-8 outlined above. The actual calculalation of the angles of arrival is done by the power method which estimates the number of sources and the DOA. To obtain the matrix in a form which is suitable for the power method it is necessary to decompose it and to obtain the eigenvalues. The Householder transformation and the QR method are used to perform the eigendecomposition. Another important modification is the use of the Cholesky decomposition to convert the $\mathbf{G}$ and $\mathbf{G}_n$ matrices into the standard form for eigendecomposition.

It is important from the implementation point of view to parallelize the algorithm so that the algorithm can be made suitable for real time processing. The algorithm was studied and all the modules which can be computed offline were identified. A flow chart of the modified and parallelized algorithm is shown in Figure 3.2.

In this case we consider a linear array of 8 sensors A segment of 64 samples is considered, which forms the single step input to the next stage of the FFT processors. As shown in Figure 3.2 a single estimation of the angles of arrival involves the processing of 64 such segments of 64 samples each. After 64 samples are collected, the next step involves the transformation of these signals from the time domain to the frequency domain by performing a 64 point FFT. The output is a symmetrical vector in the frequency domain. One side of the vector is discarded leaving a 33 element vector which is representative of the input signal at that sensor.

Collect $X_{ni}(t)$

$i = 1 \,..\, 8 \,;\, n = 1 \,..\, N$

Compute FFT for every $X_{ni}(t)$

$X_{ni}(\omega L) \quad L = L_1 \; ... \; L_{1+nf}$

Compute $X_{nj}(\omega L) \quad X^*_{nk}(\omega L)$

$j = 1 \,..\, m \quad k = j, \,..\, m \quad L = L_1 \; ... \; L_{1+nf}$

Compute Average for

$$\frac{1}{N} \sum_{n=1}^{N} X_{nj}(\omega L)\, X^*_{nk}(\omega L)$$

Form

$$\hat{K}(wl) = \frac{1}{N} \sum_{n=1}^{N} X_n(\omega L)\, X_n^H(\omega L)$$

$l = L_1, L_1 + 1 \; ... \; L_{1+nf}$

Compute

$$G = \sum_{L=L_1}^{L_{1+nf}} T(\omega L)\, \hat{K}(\omega L)\, T^H(\omega L)$$

and

$$G_n = \sum_{L=L_1}^{L_{1+nf}} T(\omega L)\, P_n(\omega L)\, T^H(\omega L)$$

(Perform Cholesky Decomposition)

Convert $GX = \lambda G X$

to the standard eigenvalue

$HY = \lambda Y$

Perform eigendecomposition

- position of (G, I)

Continued...

Continued...

Obtain

$\lambda_1 \ldots \lambda_m$

$e_1 \ldots e_m$

estimate $\hat{d}$ number of sources

estimate angle of arrival

$$P(\theta) = \frac{1}{\sum_{k=d+1}^{M} \left\| a_\theta^H e_k(\omega_c) \right\|^2} .$$

where

$$a_\theta = \begin{vmatrix} 1 \\ \omega_c \tau \\ (\omega_c \tau)^2 \\ \cdot \\ \cdot \\ \cdot \\ (\omega_c \tau)^{m-1} \end{vmatrix}$$

Figure 3.1 : Mathematical transformations in the algorithm

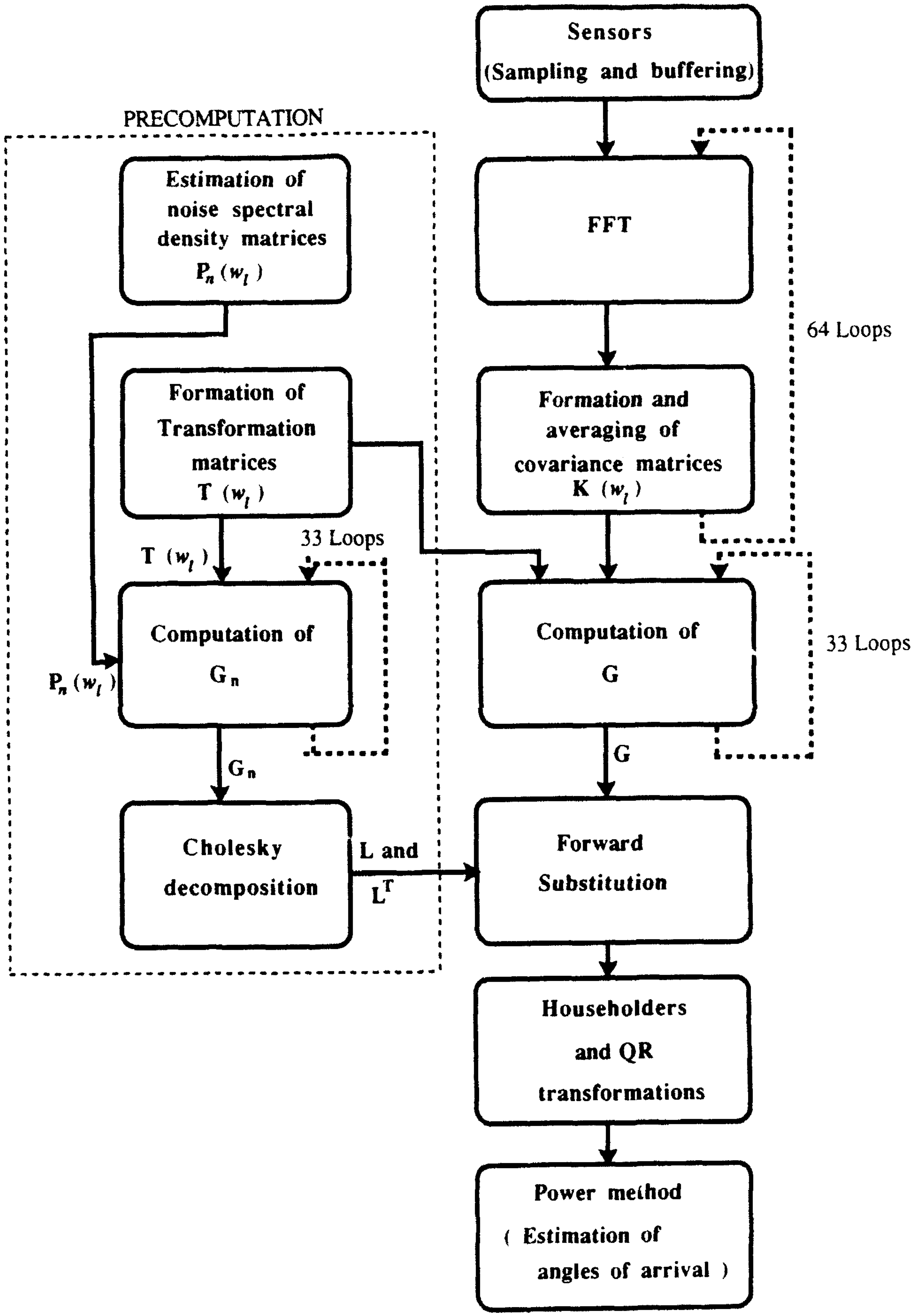


Figure 3.2 : Flowchart of modified bilinear transformation algorithm

The next block is the calculation of the covariance matrix at each frequency bin. Essentially the covariance matrix consists of the product of the frequency vector and its Hermetian which is obtained from the corresponding elements in the FFT output vectors. Hence for the 33 different narrowband frequencies there are 33 different covariance matrices independant of each other. These matrices are averaged over the 64 segments before being passed on to the next step in the algorithm which is the projection of the covariance matrices $\mathbf{K}(w_l)$ onto the single midband frequency in the spectrum to compute the **G** matrix.

The computation of the **G** matrix requires the transformation matrices $\mathbf{T}(w_l)$ which are precomputed as shown in the diagram. As seen from Equation(2 8) in the previous chapter the computation of the matrix involves the knowledge of the narrowband frequencies in the bandwidth. Given a specific problem such an estimation of the frequency bins is made by splitting the bandwidth into 32 equal parts and taking the frequencies at the boundary. With this initial assumption of the narrowband frequencies in the spectrum of the incoming signals the transformation matrices can be computed offline. This is possible because the matrices are unique for a set of frequencies and are independant of the angles of arrival of the incoming signal. Hence these invariant matrices can be stored in a ROM for a dedicated architecture and can be called up whenever they are required during the processing. However an architectural model has been developed to compute the transformation matrices on line which would enable the system to be more general purpose and allow it to run scans over different frequency ranges without the initial knowledge of their frequency components. The computation of the actual transformation matrices is outlined below following the principles explained in the previous chapter.

3.2 COMPUTATION OF TRANSFORMATION MATRICES

The transformation matrix is derived as follows:

Let **B** be a matrix constructed from the coefficients of the polynomial $p_k(z) = (1+z)^k (1-z)^{8-k}$, where k =1, 2, ... 7. K denotes the number of the row of the 8 x 8 matrix which is formed. In this case the nonsingular matrix has been computed and is shown below

$$\mathbf{B} = \begin{bmatrix} 1 & 7 & 21 & 35 & 35 & 21 & 7 & 1 \\ 1 & 5 & 9 & 5 & -5 & -9 & -5 & -1 \\ 1 & 3 & 1 & -5 & -5 & 1 & 3 & 1 \\ 1 & 1 & -3 & -3 & 3 & 3 & -1 & -1 \\ 1 & -1 & -3 & 3 & 3 & -3 & -1 & 1 \\ 1 & -3 & 1 & 5 & -5 & -1 & 3 & -1 \\ 1 & -5 & 9 & -5 & -5 & 9 & -5 & 1 \\ 1 & -7 & 21 & -35 & 35 & -21 & 7 & -1 \end{bmatrix} \qquad (3.1)$$

From this **B** matrix the transformation matrix can be computed according to Equation (2.10). For the matrix $\mathbf{D}(\frac{w_c}{w_l})$, the (k,k) th term is given by

$$d_{kk} = \left(\frac{2w_c}{jw_l}\right)^{k-1}$$

Let p denote the constant term such that

$$p = \frac{2w_c}{jw_l}$$

The transformation matrix can now be written as shown in Equation (3.2)

The matrix **T** can thus be computed and is stored in a ROM and is retrieved by each processor. The next precomputation block is the calculation of the $\mathbf{G}_n$ matrix which is the estimate of the noise spectral density that is expected to be present in the signal. The algorithm requires a previous knowledge of the noise in the system which is expressed in terms of the P_n matrices at each frequency bin. The procedure for calculating the $\mathbf{G}_n$ matrix involves two matrix multiplications and is similar to the computation of the **G** matrix from the covariance matrices. The calculations are performed 33 times, once for each frequency component and are then averaged at the midband frequency.

$$\mathbf{T}(w_l) = \begin{bmatrix} p & 7p^2 & 21p^3 & 35p^4 & 35p^5 & 21p^6 & 7p^7 & p^8 \\ p & 5p^2 & 9p^3 & 5p^4 & -5p^5 & -9p^6 & -5p^7 & -p^8 \\ p & 3p^2 & -p^3 & -5p^4 & -5p^5 & -p^6 & 3p^7 & p^8 \\ p & p^2 & -3p^3 & -3p^4 & 3p^5 & 3p^6 & -p^7 & -p^8 \\ p & -p^2 & -3p^3 & 3p^4 & 3p^5 & -3p^6 & -p^7 & p^8 \\ p & -3p^2 & p^3 & 5p^4 & -5p^5 & -p^6 & 3p^7 & -p^8 \\ p & -5p^2 & 9p^3 & -5p^4 & -5p^5 & 9p^6 & -5p^7 & p^8 \\ p & -7p^2 & 21p^3 & -35p^4 & 35p^5 & -21p^6 & 7p^7 & -p^8 \end{bmatrix} \tag{3.2}$$

The equation governing this transformation is shown below.

$$\mathbf{G_n} = \sum_{l=l_1}^{l_1+n_f} \mathbf{T}(w_l)\,\mathbf{P_n}(w_l)\,\mathbf{T}^{H}(w_l) \tag{3.3}$$

The matrix $\mathbf{G}_n$ is then stored in the ROM and accessed at the time of the Cholesky decomposition.

3.3 COMPUTATION OF G MATRIX

The **G** matrix which is the combination at the midband frequency of all the individual covariance matrices of different narrowband components requires the projection of these matrices by the transformation matrices and involves two matrix multiplications as shown in the equation below.

$$\mathbf{G}=\sum_{l=l_1}^{l_1+n_f} \mathbf{T}(w_l)\,\mathbf{K}(w_l)\,\mathbf{T}^H(w_l) \tag{3.4}$$

The process goes through 33 iterations as shown in the flowchart. Each loop involves two matrix multiplications which are done sequentially, because the input to the second operation is the output from the first. However parallelism has been achieved inside each operation as it is performed in one cycle. The computation of the **G** matrix gives the matrix pencil ($\mathbf{G}, \mathbf{G}_n$) of which $\mathbf{G}_n$ has been precomputed.

234 CHOLESKY DECOMPOSITION

The further processing of the signal requires that it be organised into a standard form so that certain standard operations of matrix algebra like the eigendecomposition can be performed. The algebraic manipulations which are performed to achieve the objective are described below.

$\mathbf{G}_n$ and **G** are two matrices which need to be put in the standard form such that

$$\mathbf{G}\,\mathbf{X} = \lambda\,\mathbf{G}_n\,\mathbf{X} \tag{3.5}$$

where λ = the eigenvalues of **G**

$$\mathbf{X} = \text{the eigenvector matrix of } \mathbf{G}_{\prime\prime} \text{ and } \mathbf{G}$$

Decomposing $\mathbf{G}_{\prime\prime}$ into

$$\mathbf{G}_{\prime\prime} = \mathbf{L}\,\mathbf{L}^{\mathbf{T}} \tag{3.6}$$

and substituting $\mathbf{G}_{\prime\prime}$ in the equation and multiplying both sides by $\mathbf{L}^{-1}$ gives

$$\mathbf{L}^{-1}\,\mathbf{G}\,\mathbf{L}^{-\mathbf{T}}.\,\mathbf{L}^{\mathbf{T}}\,\mathbf{X} = \lambda\ \mathbf{L}^{-1}\,\mathbf{L}\,\mathbf{L}^{\mathbf{T}}\,\mathbf{X}$$

Defining $\quad \mathbf{L}^{-1}\,\mathbf{G}\,\mathbf{L}^{-\mathbf{T}} = \mathbf{H}$ and $\mathbf{L}^{\mathbf{T}}\,\mathbf{X} = \mathbf{Y}$ (3.7)

The standard form required for eigendecomposition can be written as

$$\mathbf{H}\,\mathbf{Y} = \lambda\mathbf{Y} \tag{3.8}$$

The decomposition $\mathbf{G}_{\prime\prime} = \mathbf{L}\,\mathbf{L}^{\mathbf{T}}$ is obtained by doing the Cholesky decomposition which is the next step in the algorithm as shown in the flowchart.

The flowchart of the Cholesky decomposition is shown in Figure 3.3. The objective of reducing to a lower triangular matrix is achieved by computing the elements below the diagonal according to the equation

$$a_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} a_{ij}a_{kj}}{a_{ii}} \tag{3.9}$$

The diagonal elements are however computed by the formula

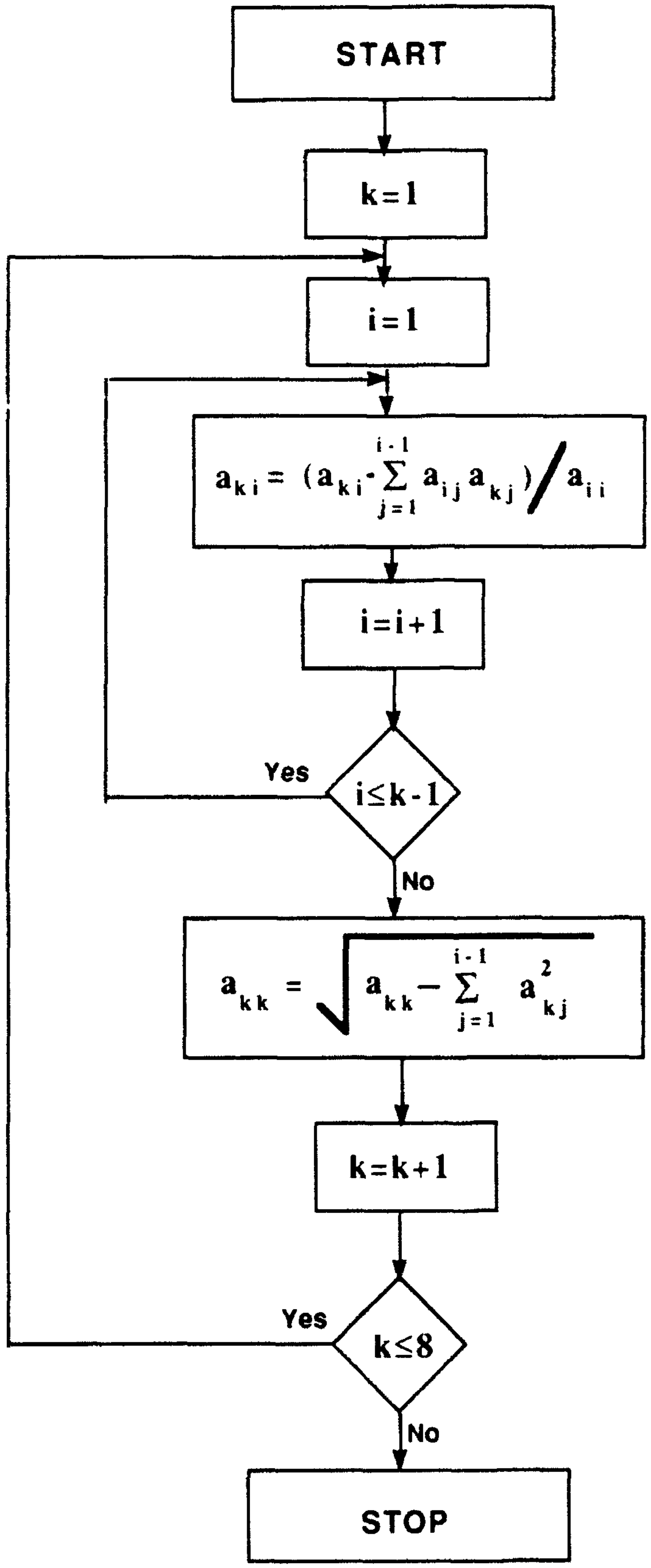


Figure 3.3 Flowchart for Cholesky Decomposition

$$a_{kk} = \sqrt{a_{kk} - \sum_{j=1}^{k-1} a_{kj}^2} \tag{3.10}$$

Once the lower triangular matrix **L** has been computed the transpose $\mathbf{L}^T$ can be obtained.. The next step is to obtain the two matrices **H** and **Y**. This part needs the calculation of the inverse of the lower triangular matrix L as is seen from Equation(3.7). This computation is both time consuming and complex especially for real time applications. The ultimate objective is not to calculate the inverse and to circumvent this requirement, a simple algebraic manipulation is described below:

Assuming a matrix **W** such that

$$\mathbf{L\,W = G} \tag{3.11}$$

we have

$$\mathbf{W = L^{-1}\,G}$$

Taking the transpose and premultiplying both sides of Equation (3.11) by $\mathbf{L}^{-1}$ gives

$$\mathbf{L^{-1}\,W} = \mathbf{L^{-1}\,(L^{-1}G)^T}$$

$$= \mathbf{L^{-1}\,G^T(L^{-1})^T}$$

$$= \mathbf{L^{-1}\,G(L^{-1})^T} \qquad \text{(as } \mathbf{G} \text{ is Hermitian)}$$

$$= \mathbf{H}$$

Hence

$$\mathbf{L\,H = W^T} \tag{3.12}$$

Considering the two Equations (3.11) and (3.12) it can be seen that the problem of computing the inverse is now reduced to the computation of the **H** matrix by two forward substitution operations. First the matrix **W** is computed from the Equation (3.11) as the other two matrices are known. Then it is transposed, which is a simple routing exercise in the architecture and the result in Equation (3.12) is used to compute the **H** matrix.The computation of **Y** also follows the same procedure.

The resultant matrices are now in one particular frequency and can be treated as a narrowband case. The two most common methods that can be applied are the MUSIC and ESPRIT algorithms. In this case the MUSIC algorithm is applied. First the Householders and QR transformations are performed to reduce the dense matrix into a diagonal one and then the power method is used to compute the angles of arrival.

`CHAPTER 4

System Architecture and Design

4.1 INTRODUCTION

In the hardware implementation of the proposed algorithm it is necessary to consider the tradeoffs between the timing requirements and the number of processors in each stage. Though parallelization and pipelining of most tasks in the process is possible this would require a large number of processing elements which are not really necessary as far as the timing requirements are concerned. Since the processing speed is going to be determined by the sampling rate at the sensors which is not very high, the basic system is configured for a system with 8 sensors. Therefore the architecture is designed such that each stage has 8 blocks of processors with the processing done in such a manner that the flow of data between processors is minimized. The system can thus be configured for a different number of sensors with minimal alteration at the architectural level.

The overall block diagram of the architecture is shown in Figure 4.1. The first part shows the sensors and the buffering stage. To obtain one segment of data for further computations each of the sensors sample 64 time delayed elements. The input to the FFT processors is therefore a 64 element vector and a buffering stage is provided to store and accumulate the data. The buffer has a control mechanism to coordinate data flow from the FFT processors. The data is transferred to all the processors simultaneously a sample at a time. A sample consists of a complex element with data being represented in signed 8 bit numbers for the real and imaginary parts.

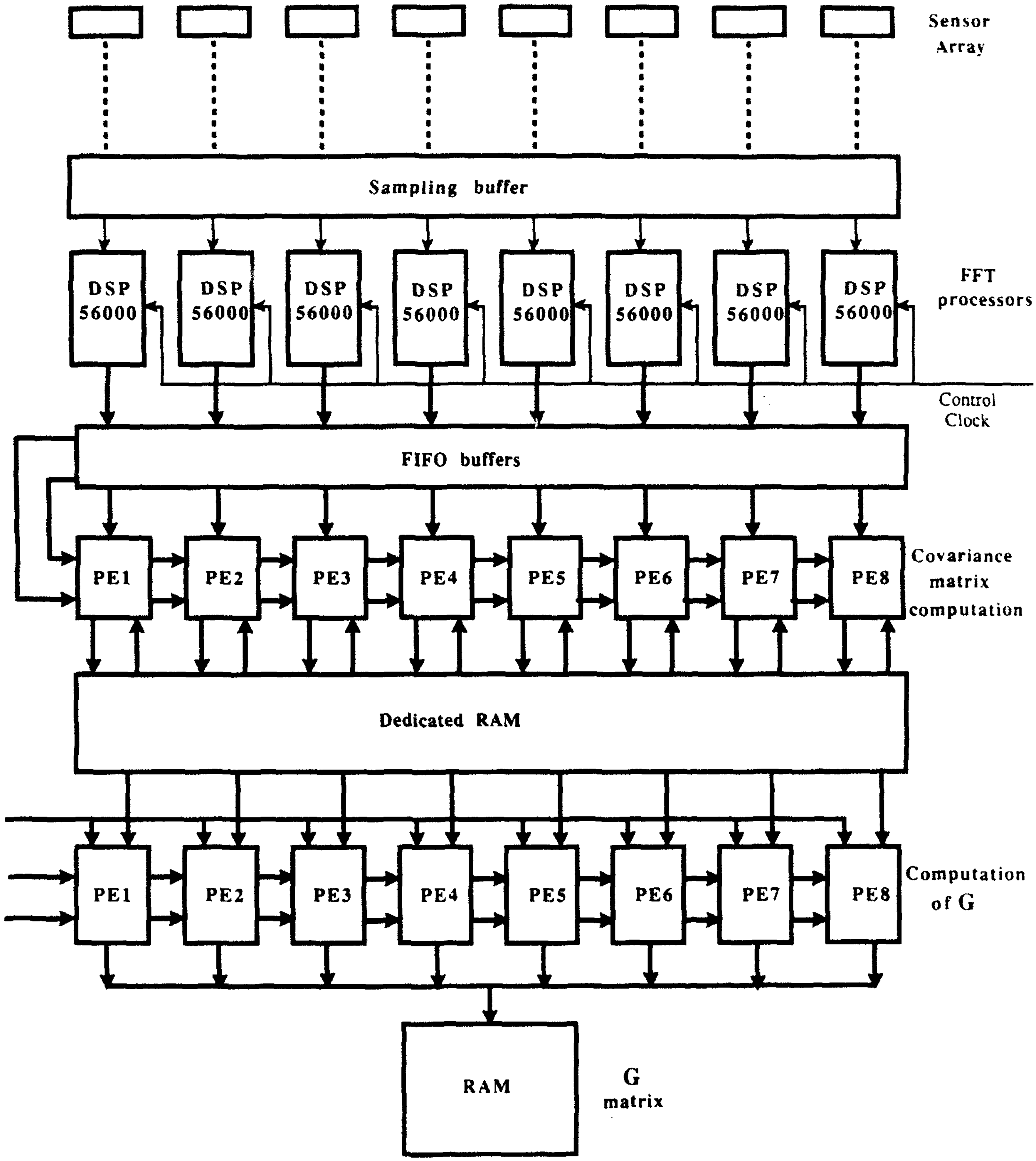


Figure 4.1: Overall system architecture till computation of G

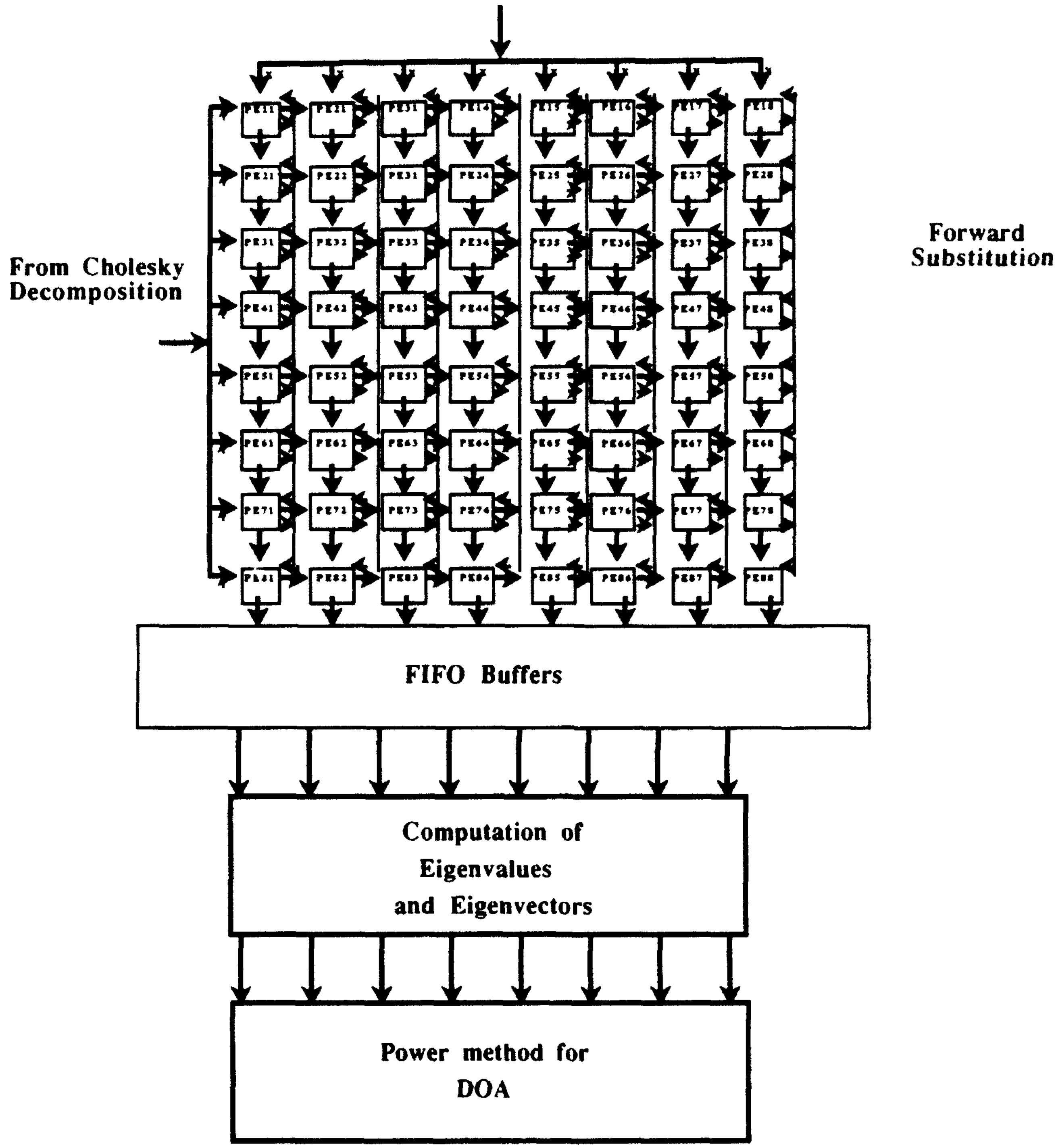


Figure 4.1: Overall system architecture for the bilinear transformation algorithm

The next stage consists of the FFT processors. In this algorithm the computation of the angles of arrival is done in the frequency domain so the first operation that is performed on the incoming data is the Fourier transform. The DSP 56000 [16] chip is proposed for the calculation of the FFT for the data from each sensor. From the specifications of the chip it has been calculated that it can perform the 64 point FFT in about 120 μs, which is acceptable for this algorithm. The output from the FFT processors is a 64 element vector in the frequency domain. But the components of the vector are symmetrical and hence for computation purposes only one side of the spectral elements is considered. The data reduces to a vector of 33 elements which is used to compute the covariance matrices. The architecture for the covariance matrix attempts to keep the symmetry of using 8 processors for each stage. A set of FIFO buffers is used between each set of processors to store the results from the FFT operation. A clock signal as shown in the figure is used to retrieve the data from the buffers in a synchronous mode which is necessary for the input to the covariance matrix processors.

4.2 THE COVARIANCE MATRIX PROCESSORS

The computation of the covariance matrix at each frequency bin essentially involves the multiplication of two 8 element vectors. These correspond to the frequency component at each of the sensors and are indicative of the change in the observed signal between the sensors. Figure 4.2 shows a more detailed diagram of the architecture for the computation of the covariance matrix. As shown in Figure 4.2 this stage consists of 8 processors each of which is used to compute one column of the covariance matrix. The flowchart in Figure 4.3 shows the various steps involved in the calculation of the 33 matrices. Figure 4.4 shows a more detailed diagram of the processors in the covariance stage.

Basically the computation of the covariance matrix involves the multiplication of a vector with its transpose resulting in a square matrix whose

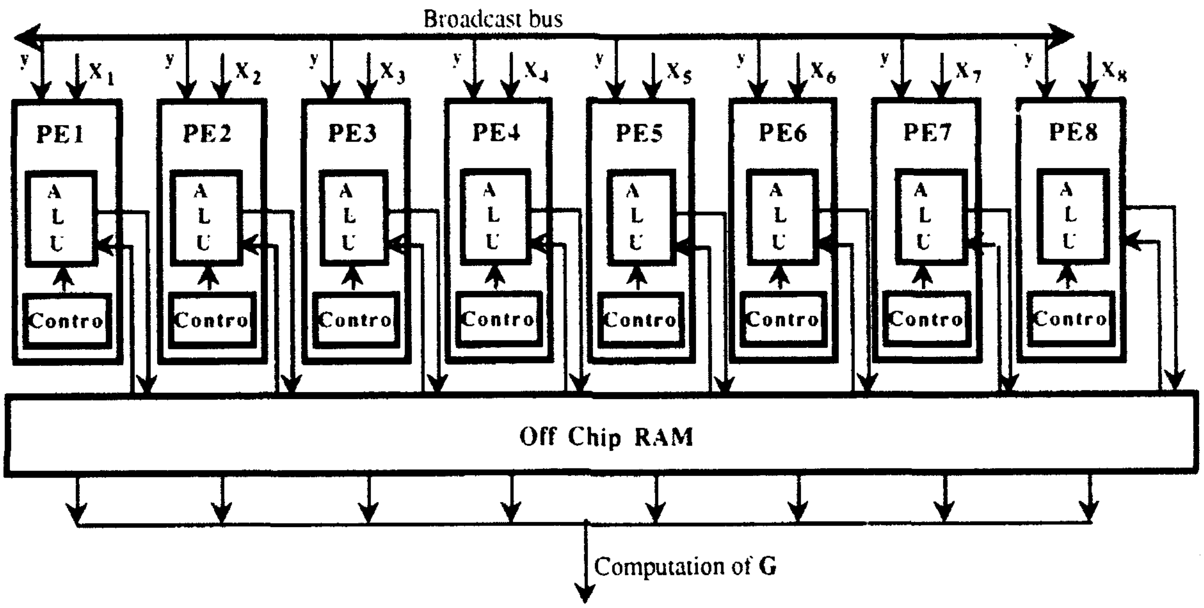


Figure 4.2: Architecture for the computation of covariance matrces

dimensions are the size of the vector. In this case the number of elements in the vector is 8, which gives a 8x8 covariance matrix. This also permits the mapping of the computation process upon an array of 8 processors, each of which calculates one column of the resultant matrix. Each column is formed by the product of that particular element with the whole vector. For example the third column (which is computed by the third PE), is formed by the product of the third element with the entire column. Hence the inputs to the third processor will be the third element (**X**) and the vector (**Y_1 - Y_8**).

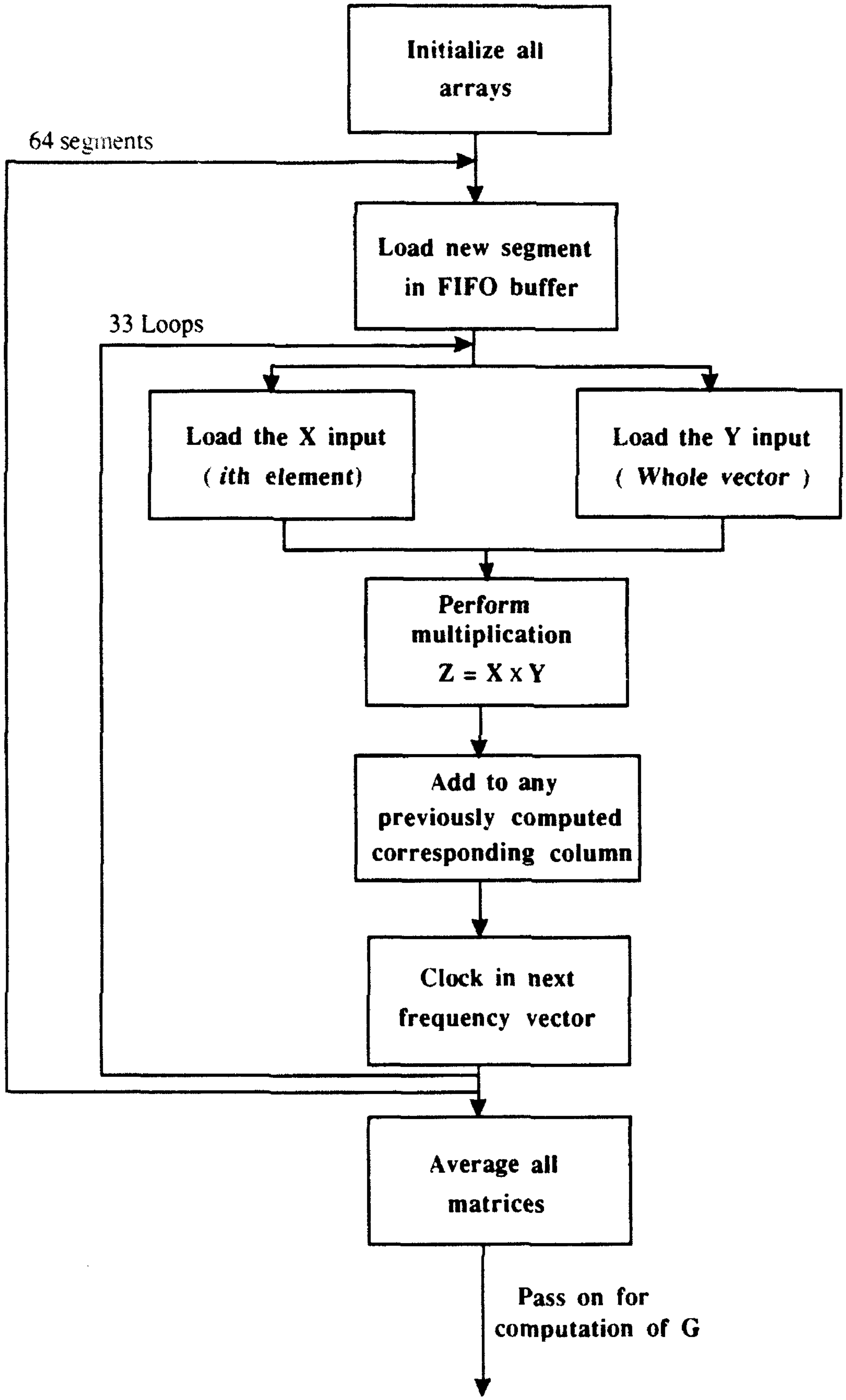


Figure 4.3 - Flowchart for computation of covariance matrices

These elements are obtained from the FIFO buffers in which the output from the FFT processors are stored. The loading of these elements can be achieved in parallel with a multiplexed bus which will route the data from a buffer to a single PE (**X** input) and a broadcast bus which will put the data to all the processors (Y_1 - Y_8 inputs).

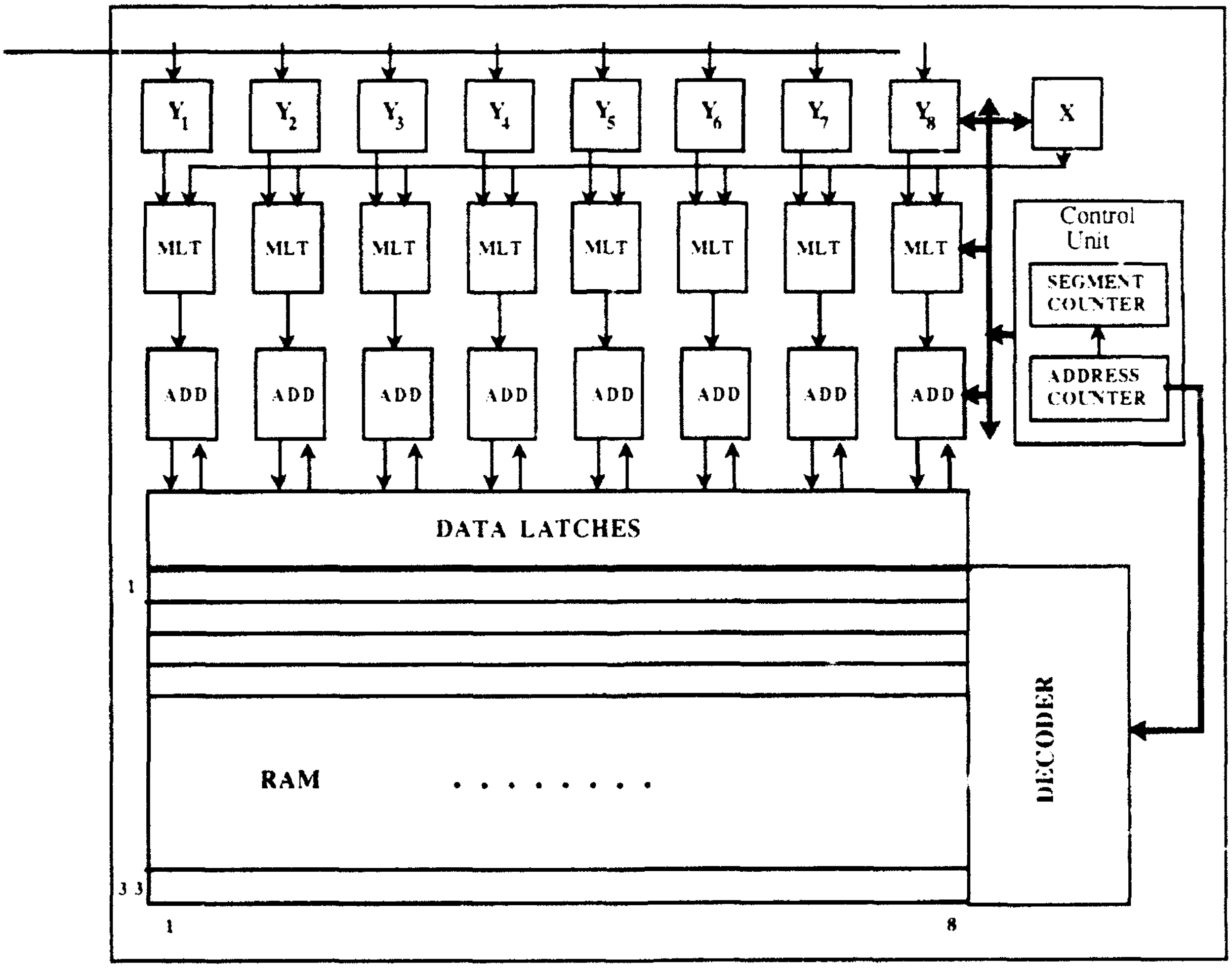


Figure 4.4 Processing Element for covariance matrix stage

As seen from Figure 4.4, once the data is latched in to the buffers inside the PE, it is passed on to a multiplier. In the block diagram each MLT is of a complex number multiplying unit consisting of four multipliers, one adder and a subtractor. The two multiplier inputs are the **X** value and the corresponding Y value. Once the product is computed, it is passed on to an accumulator, which adds the incoming value to one that has been computed from the previous segment. This previous value is stored in a local or off the chip RAM as shown in Figure 4.4 and can be retrieved as follows.

The control unit inside the PE basically has the function of supplying the various signals which would enable the correct data to be retrieved from the local RAM during the arithmetic operations. An address counter which runs from 0 to 32 will generate the address which is needed to retrieve the proper vector from the RAM. The decoder takes the signal from the counter and enables a particular row which contains the vector corresponding to that frequency. The particular vector is put on the data latches from where it goes to the adder. This completes the read cycle from the memory. Once the addition is done, the data is now written back into the latch overwriting the data which had been previously stored. A write cycle is executed and the acccumulated result is written back into the same memory cells. The address is held valid till the write operation is completed. The counter is now incremented which takes the whole operation into the next cycle. Once the counter completes 33 cycles it is reset and a pulse is sent to the segment counter which is incremented. The segment counter is set to run from 0 to 63 and is used to indicate the end of a frame.

The memory is organized into an array of 8 x 33 cells. Each cell is capable of storing one element of the vector. The word length is such that 8 elements can be accessed in one cycle on parallel data buses. The addresses run from 0 - 32 for the 33 vectors that are stored. Once the computations have been performed for one frame they are averaged, and passed on for the computation of G.

4.3 PROCESSOR FOR THE COMPUTATION OF G MATRIX

The computation of the **G** matrix reduces the 33 frequency matrices into one single matrix. An important aspect to note is that this computation is required to be done only once every frame, i.e. every 64 segments. The architecture is very similar to the one used for the covariance matrix computation except that the operations are now matrix based instead of being vector based. This calls for a slight change in the memory requirements and the operations in the computation. As shown in Figure 4.1 the architecture consists of an array of 8 processors. Each processor is used to compute one column of the resultant matrix.

The formation of the **G** matrix involves two matrix multiplications, which are used to project the 33 frequency matrices into a single combined matrix at the central frequency according to the following equation

$$\mathbf{G} \quad = \quad \mathbf{T}(w_l)\,\mathbf{K}(w_l)\,\mathbf{T}^{\mathbf{H}}(w_l)$$

As the matrices are 8x8, the operations are mapped in an 8 processor array as shown in Figure 4.1. Each processor computes one column of the resultant matrix. The data routing is a bit more complex this time because the operands are matrices which need to be loaded into each processor. To simplify this problem the architecture is configured in such a way that only one column needs to be unique to each processsor. In this case it would be the ith column of the $\mathbf{T}^{\mathbf{H}}(w_l)$ matrix going to the ith processing element. The rest of the data (i.e. the $\mathbf{T}(w_l)$ and the $\mathbf{K}(w_l)$ matrices are broadcast simultaneously to all the processors during the computation. The $\mathbf{T}(w_l)$ and the $\mathbf{T}^{\mathbf{H}}(w_l)$ matrices can be precomputed, as they are independent of the angles of arrival and are dependent only on the frequency spectrum, which is known a priori. Hence they can be stored in an external ROM and retrieved whenever required. The computation of a column of **G** at each processor can be done by two consecutive multiplications of an 8x8 matrix with an 8x1 vector each of which results in an 8x1 column vector. The first operation is multiplying the covariance matrix $\mathbf{K}(w_l)$ to the ith column of the $\mathbf{T}^H(w_l)$ matrix, which gives the ith column of the $\mathbf{K}(w_l)\,\mathbf{T}^H(w_l)$ matrix. Next the $\mathbf{T}(w_l)$ matrix is multiplied to the previous result which gives the ith column of the **G** matrix at the ith processor.

A flowchart of the process of computation of the **G** matrix is shown in Figure 4.5. The algorithm has been parallelized so that the processor can execute nonsequential operations at the same time. The first operation is the loading of the two input vectors, which are done simultaneously. The next set of operations involve the parallel multiplication of the vector elements. At the same time the next row of the $\mathbf{K}(w_l)$ matrix can be loaded into the input latch. Also from the second loop onwards the results can be accumulated. Next the eight elements are added to give the innerproduct which is one element of the column. This repeats for eight loops to compute all the elements of the 8x1 column.

Similarly the second matrix multiplication is performed except that this time the **X** input is the resulting column of the first multiplication and the **Y** input is the row of the $\mathbf{T}(w_l)$ matrix. This operation is repeated eight times to compute the **G** matrix for the first frequency bin. The process then runs through 33 iterations for the 33 frequencies. The values are averaged and the final **G** matrix is calculated.

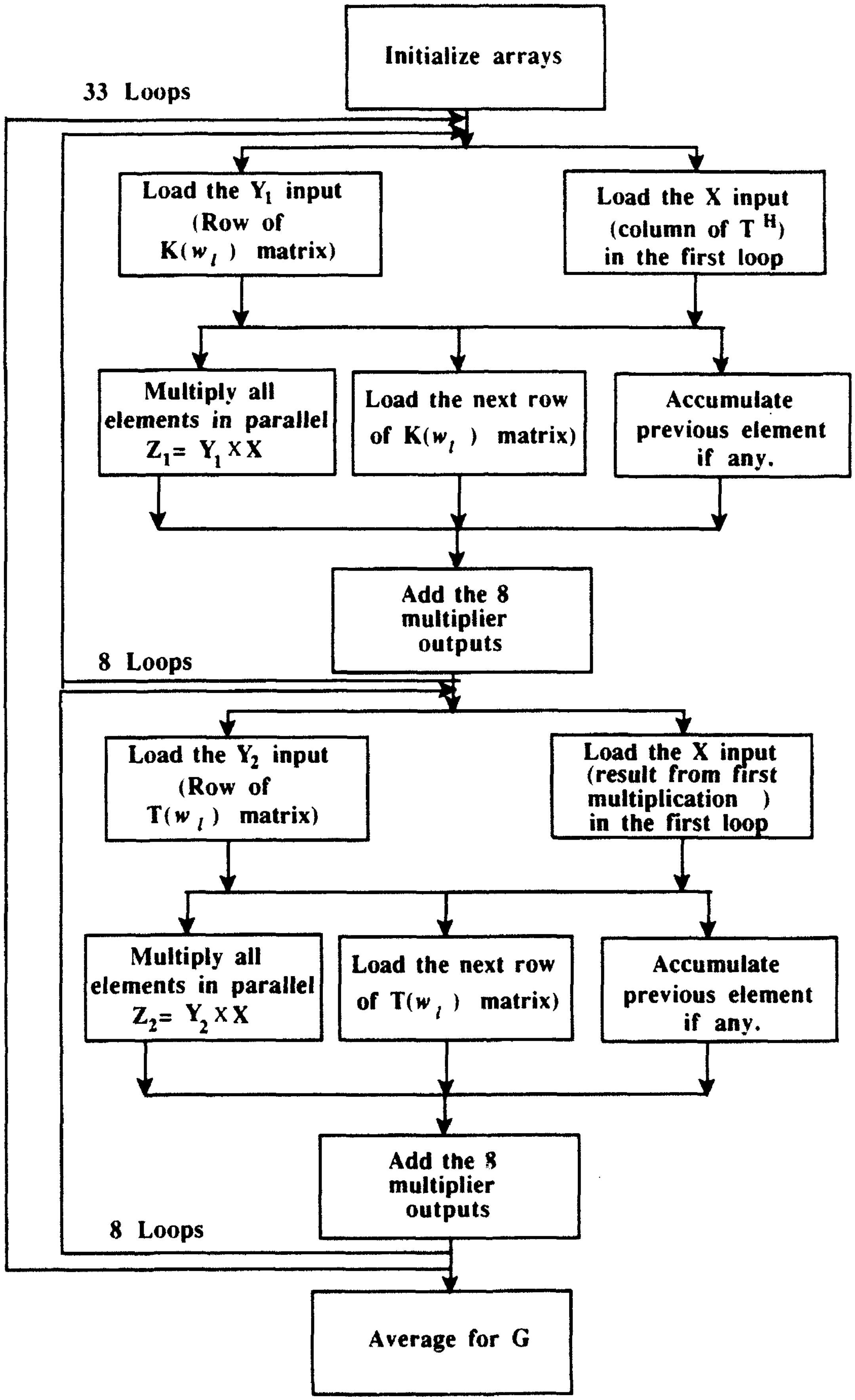


Figure 4.5 Flowchart for the computation of G matrix

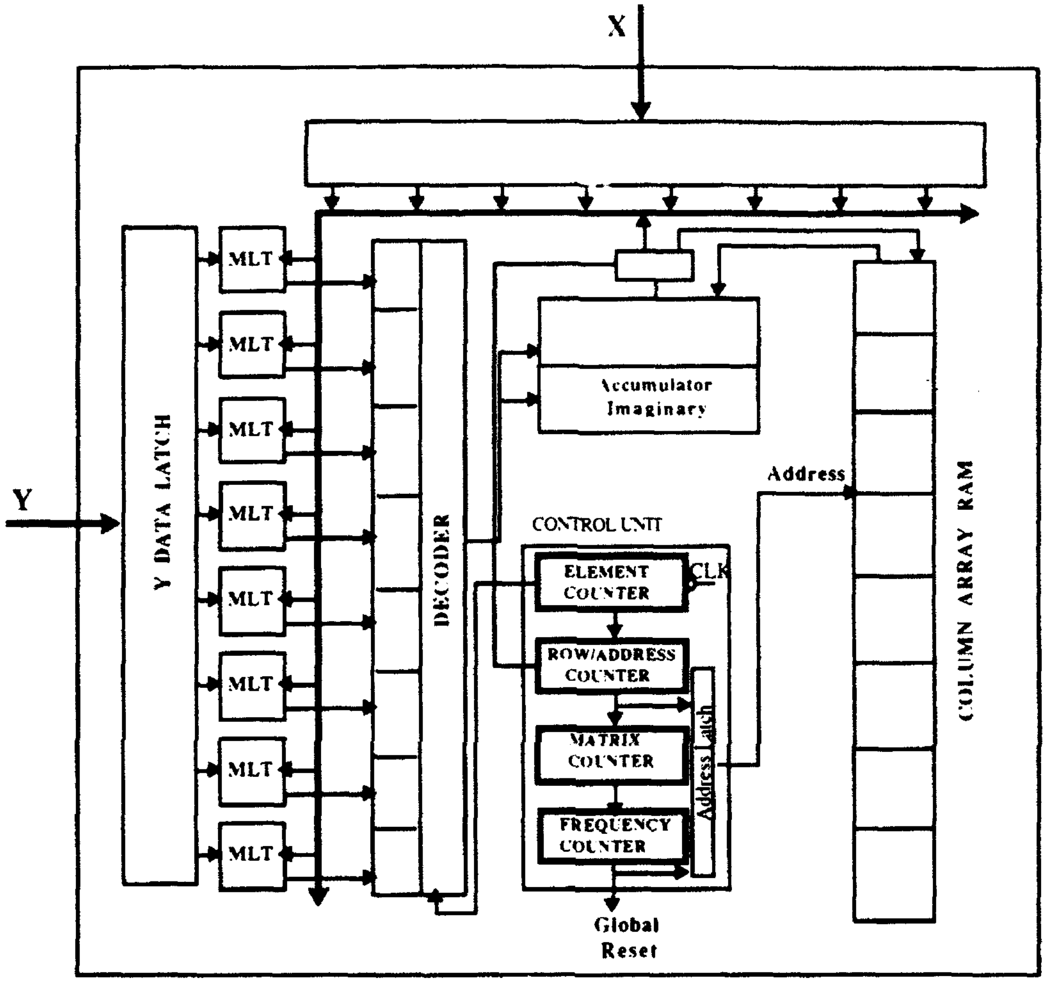


Figure 4.6 Processing Element for the computation of G matrix

The internal block diagram of the PE used for the calculation of the **G** matrix is shown in Figure 4.6. The **X** input is the *i*th column vector of $\mathbf{T}^{\mathbf{H}}(w_l)$. For the fourth processor the input would consist of the fourth column of the $\mathbf{T}^{\mathbf{H}}(w_l)$ matrix. The loading can be done in parallel, to all the processors. The other input consists of the $\mathbf{K}(w_l)$ and the $\mathbf{T}(w_l)$ matrices. The sequence of operations is shown in the flowchart and has been explained above. The eight multipliers perform the eight complex multiplications required to form the innerproduct in parallel. The results are fed through a multiplexer to an adder which sums them up, and stores the result in the memory array which can be retrieved for later processing. The new row for the next loop is loaded into the data latch when the multiplications are being performed. Once the process goes into the second frequency the adder also has to retrieve the data from the array and add to the newly computed value. This operation is performed by first reading the data from the RAM, adding it and writing the result back into the same memory location.

The control unit essentially consists of four counters which are used to keep track of various operations being performed. The first counter is the element counter which upcounts to eight and is used to control the innerproduct computation. It enables the latches, which load the data from the appropriate mulltiplier in to the adder. Once the element counter counts eight, it is reset and a pulse is sent to the row/address counter which is incremented. The row/address counter also counts to eight and keeps track of the row of the input matrix that is being loaded. This counter also provides the address for the RAM to store and retrieve the data. The third counter is a matrix counter which counts the matrix multiplications. It is a simple inverter and specifies the first or second multiplication. This is complemented every time the row/address counter is reset. The output of the matrix control is used to load the appropriate matrix into the processor. The last counter is the frequency counter which counts upto thirty three frequency bins. The outputs from the last two counters are basically used to retrieve the appropriate data from the buffers. Once the **G** values are computed for all the frequency bins, the processor then averages the column to give the value of the column of **G**. The whole matrix is obtained from the columns from the eight processors.

4.4 COMPUTATION G_n

This DOA algorithm requires the knowledge of the noise spectra in the signal which is finally expresssed in the form of the $\mathbf{G_n}$ matrix. The $\mathbf{G_n}$ matrix can be computed similar to the **G** matrix except that the signal vectors are replaced by sampled signals which do not have any wavefronts from the objects in them. i.e. they are representative of the medium only. This operation needs to be performed only for updating the $\mathbf{G_n}$ matrix. As explained in the previous section there is one operation which is performed on the $\mathbf{G_n}$ matrix which is not performed on the **G** matrix, which is the Cholesky Decomposition. This operation is required to put the two matrices into the standard form for further processing. The Cholesky decomposition can be carried out effectively offline from the main processing stream, and the result fed back online whenever the need arises. The architecture for this operation is explained in Section 4.6.

4.5 FORWARD SUBSTITUTION

As explained in the previous section the **G** matrix needs to be decomposed into a standard form. This transformation is accomplished by performing two forward substitution operations as explained in Section 3.4. The steps in the forward substitution are more complex than the previous stages because the operation involves a series of multiply and accumulate steps to calculate each element. Hence to reduce the complexity of the PEs, a systolic architecture is adopted for this stage. Figure 4.7 shows a completely parallel and pipelined architecture for this operation.

The stage consists of an array of 8x8 processors each of which computes one element in the matrix. A detailed figure of a typical processing

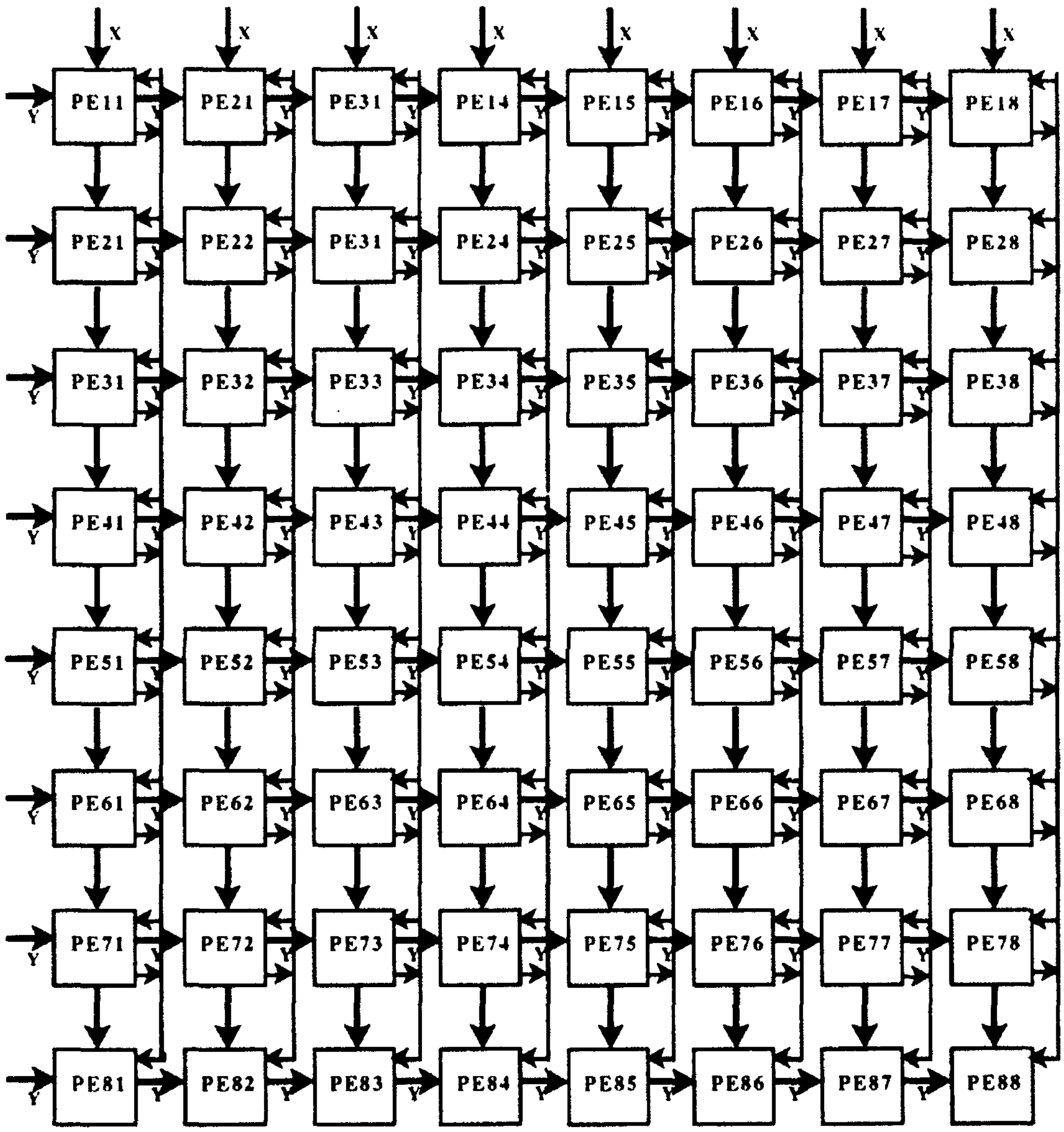


Figure 4.7 Fully pipelined and parallel architecture for the Forward Substitution operation

cell is shown in Figure 4.8. The **Y** input in this case is the particular column of the lower triangular matrix **L** and the **X** input is the corresponding element from the **G** matrix. As before the **X** input is unique to the PE while the **Y** input is broadcast to all the PEs in that column. All the outputs are transmitted downwards for further processing. In the first cycle the first row

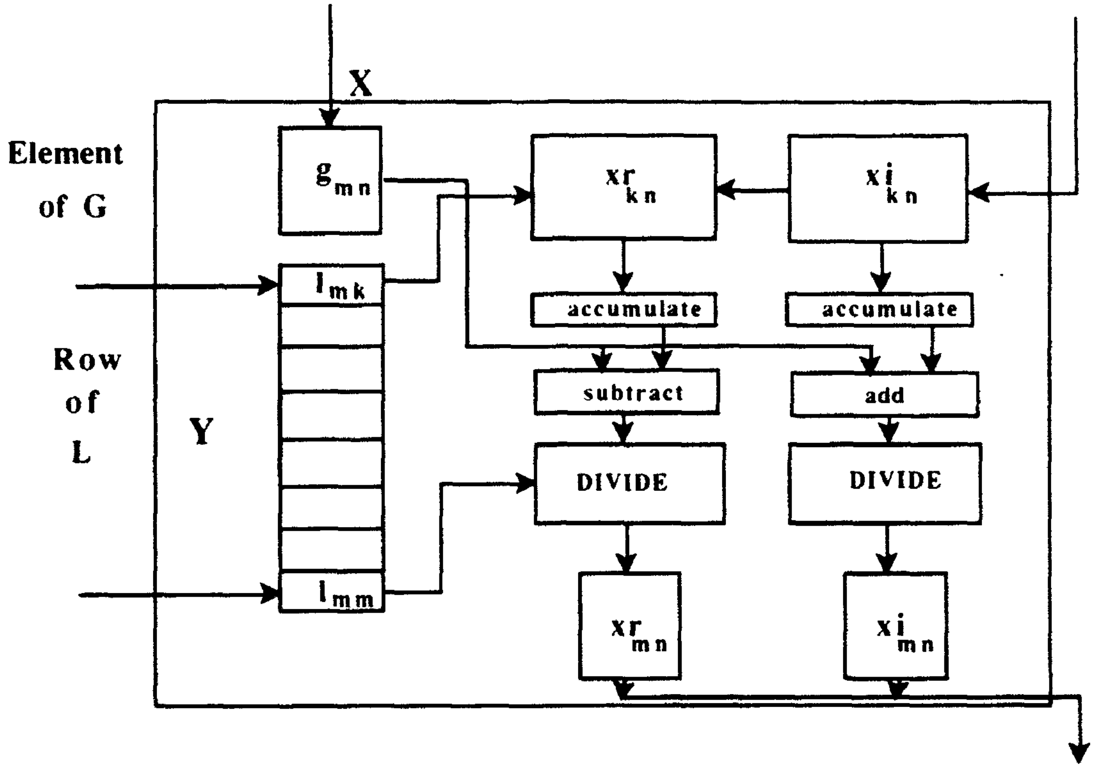


Figure 4.8 Processing Element used in the Forward Substitution stage

elements are computed. The result is broadcast to all the processors directly beneath it. From the second cycle onwards the processsors beneath the row of that particular operation, will be active while those which have already calculated their corresonding elements are inactive. The whole process of calculation of the result takes eight cycles. After it is done, the next set of data is loaded to compute **H** for the standardization.

4.6 CHOLESKY DECOMPOSITION

The flowchart for the Cholesky decomposition shows the various sequence of steps which the processors have to perform. Figure 4.9 shows the array which is used for Cholesky decomposition of the G_n matrix. The triangular array is loaded into the processors with each element going to its corresponding processor. The processors along the diagonal are different from the processors below it as they have different computations to perform. The computation

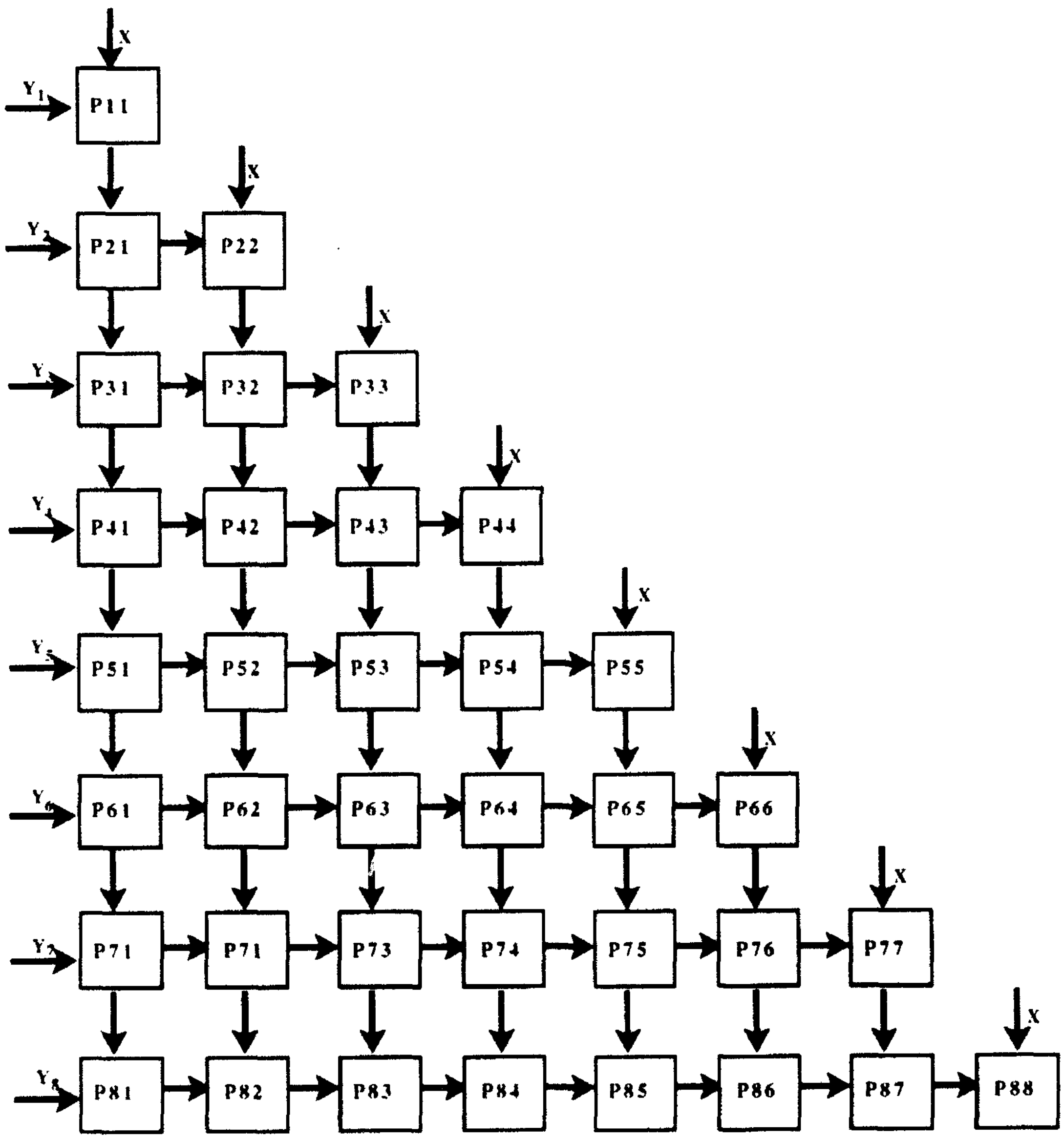


Figure 4.9Architecture for Cholesky decomposition

process takes eight cycles during each of which one column of the resultant L matrix is computed.

The initial inputs are the individual elements of the matrix. Unlike the previous processes, the input to the processors in the Cholesky decomposition change during every cycle depending upon the number of the column that is

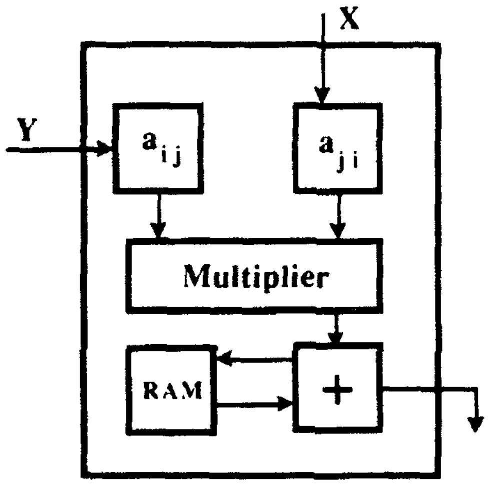


Figure 4.10 : Processing Element for Cholesky Decomposition

being computed.The **X** input to a PE will be the above diagonal elements of the corresponding column while the **Y** inputs are the corresponding elements from the same row. The results are accumulated after every multiplication. For example when the sixth row is being computed, there will be five multiplications and additions before the final subtraction and division. The accumulated value is subtracted from the original element value and then divided by the column's diagonal element. The equation for computing the subdiagonal element is

$$a_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} a_{ij} a_{kj}}{a_{ii}}$$

and the diagonal elements are computed by the equation

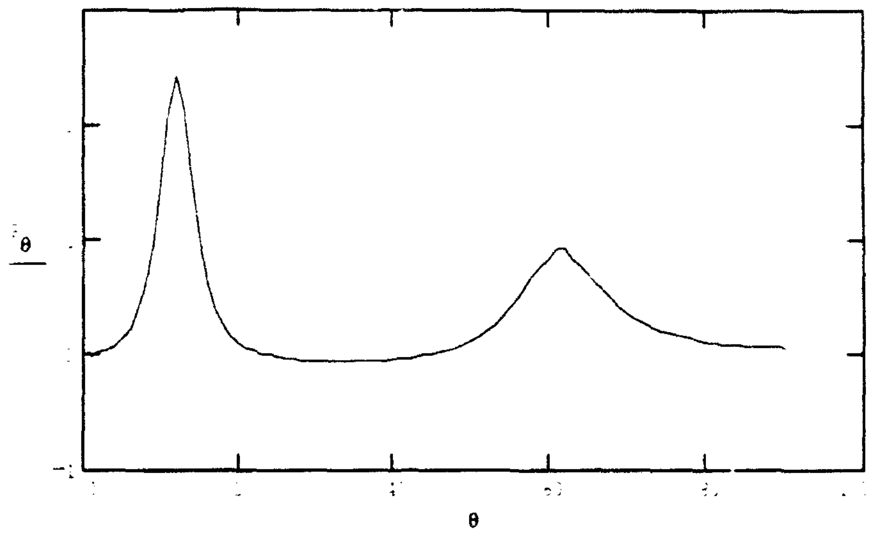
$$a_{kk} = \sqrt{a_{kk} - \sum_{j=1}^{k-1} a_{kj}^2}$$

Hence the PEs on the diagonal have a slightly different function to perform than the PEs below the diagonal and hence are a little different.

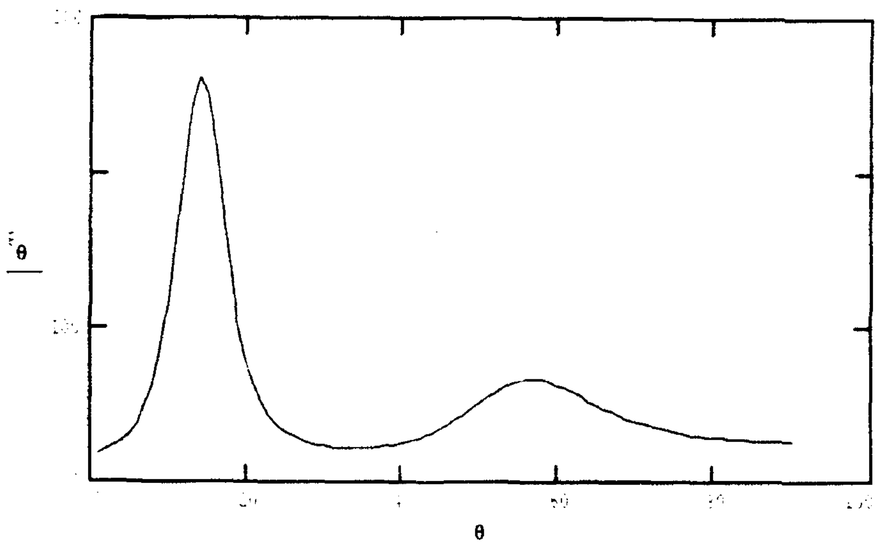
Once the 33 spectral matrices have been combined at a single frequency then the computation can be carried out by the Householder/QR transformations and the Power method.

4.6 PROCESSOR WORDSIZE VERIFICATION

One major obstacle in the design of the processor is the estimation of the number of bits that a complex number needs to be represented. The important consideration is that the algorithm must resolve the number of sources without the loss of too much resolution. For this purpose the algorithm was simulated by assuming eight bits for the real and imaginary parts. During the simulation the sensor signals were quantized to 8 bit numbers and the procedure was carried out. The power method estimated the angles of arrival with the required accuracy and resolution. The plots for the DOA for the quantized and unquantized methods are shown in Figure 4.11. It can be seen that the scaling down of the signal mainly has the effect of reducing the absolute value of the DOA power estimation but does not affect the ability of the algorithm to discriminate between sources.



DOA for 8 bit quantized signal



DOA for unquantized signal

Figure 4.11 : Direction of Arrival Estimation using quantized and unquantized data

CHAPTER 5

A combined covariance matrix processor

5.1 INTRODUCTION

A common step in most algorithms used for the estimation of DOA is the computation of the covariance matrix from the incoming signals. This is generally the first preprocessing step which generates a correlation function from the data that is collected at the sensors. From the VLSI implementation point of view it is very appealing to design a combined covariance matrix processor which will be programmable and can be used for both narrowband and broadband algorithms. Such a combined processor has the advantage of being very cost effective and opens avenues to design a configurable system.

In this work three such algorithms which are very appropriate for the development of a dedicated system are considered and a combined covariance matrix processor is developed for them. The design of such a processor is possible because the basic computations required in this stage are complex multiplications, accumulations and averaging, which are common to the three methods considered in this work. One algorithm is the bilinear transformation algorithm which has been described in the previous chapters. The other two are the narrowband MUSIC algorithm [17] and a broadband BASS-ALE method [18]. The processor is designed to be compatible with eight sensors and eight processor system described for the bilinear transformation method and shown in Figure 4.2. Eight processors in the system are placed on a processing board with each processor having its dedicated memory as shown in Figure 5.1. In this work the

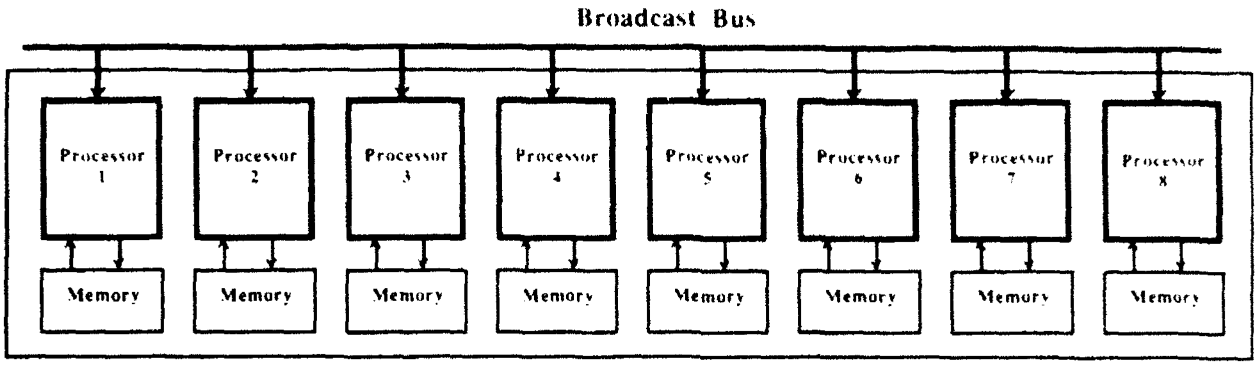


Figure 5.1 : Processing board for the computation of covariance matrix

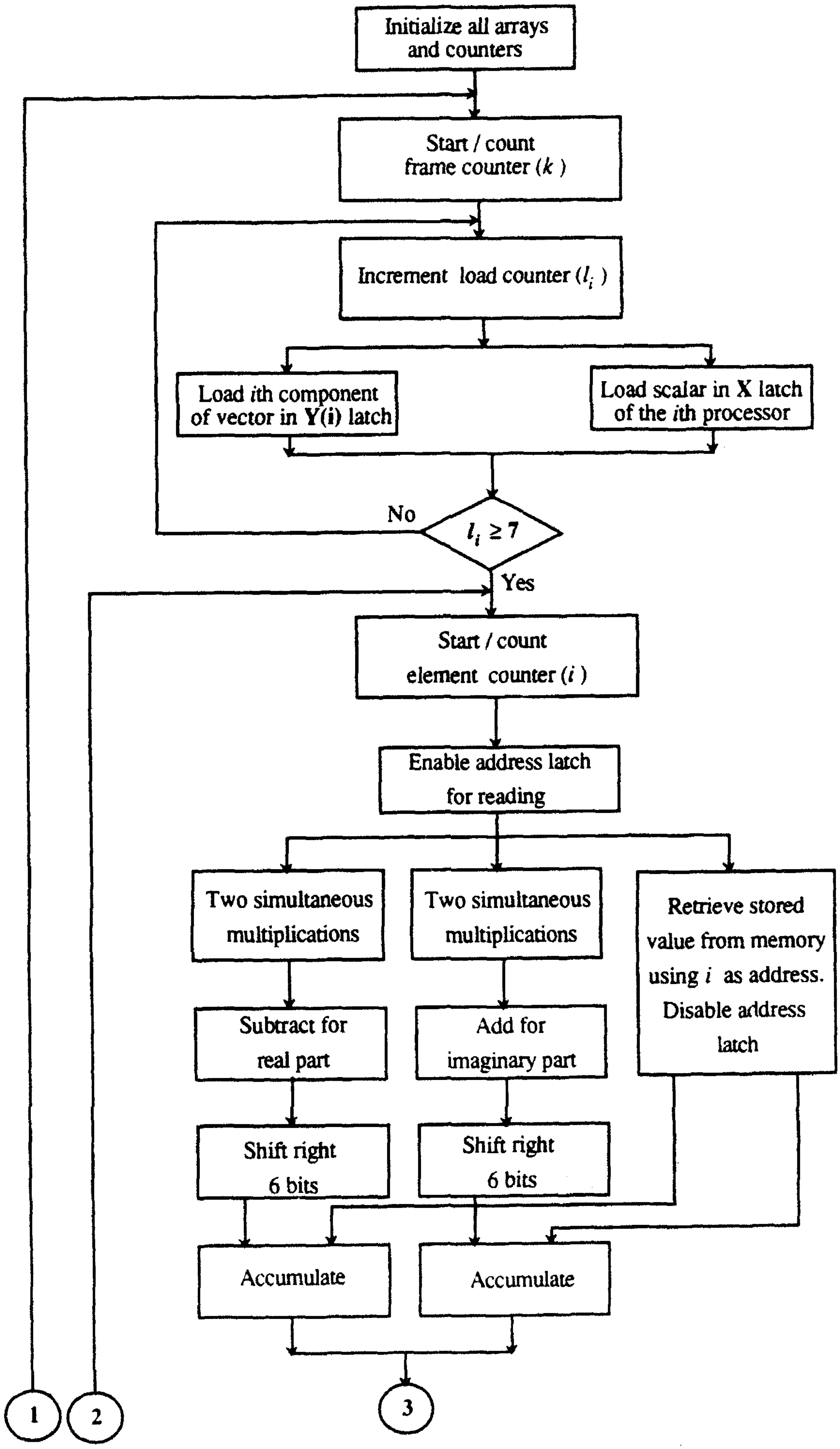
design and implementation of an ASIC chip for the processor is carried out. The processing board can be completed by using commercially available components for the memories. Though the above two algorithms are not discussed in detail the procedure involved in generating the covariance matrices is explained.

5.2 COVARIANCE MATRIX COMPUTATION FOR MUSIC ALGORITHM

In the case of the narrowband MUSIC algorithm the covariance matrix generation is the first step after the sensor stage. To form one matrix we need a vector of 8 elements from the sensors which are sampled simultaneously. The covariance matrix is therefore an 8 X 8 matrix formed similar to the bilinear transformation case. The matrix can be computed by the processing board shown in Figure 5.1. Each processor will compute one column of the matrix. To do this each processor has to multiply the 8 element signal vector by a single scalar which is the element in the vector corresponding to that particular processor. For example the sixth processor in the array will multiply the vector by the sixth element in it. The initial data flow requirements can thus be stated as follows. The complete vector is broadcast to all the processors. The scalar element corresponding to each processor is individually routed to it. For the narrowband case for each computation of the DOA a total of 4096 such vector samples are collected. The covariance matrix is computed for each vector and the final matrix is obtained after taking the the average of 4096 computations

The flow chart for the case of the MUSIC algorithm is shown in the Figure 5.2. First of all the whole system is reset using a global reset signal which clears all memory arrays and latches and initializes them to zero. The next step is to enable the frame counter (k) which counts the number of frames. For each new frame the PE needs to load the incoming sampled data. The control of the loading operation is handled by an external load counter (l_i) which synchronizes the loading of the data in all the processors. The loading operation is done over eight cycles during which one element is loaded for every cycle. In the first cycle the first element is loaded into the Y(1) latch of all processors and the X latch of the first processor. The latches in the processors are enabled by a decoder which is addressed by the 3 bit load counter. Once the loading is complete the arithmetic operations are started.

The next operation is the enabling of the element counter which will count eight elements of the resultant covariance matrix. To complete the arithmetic operation to generate the covariance matrix for complex numbers, it is necessary to perform four multiplications, an addition and a subtraction for each element. Apart from this there is an accumulation operation which is used to average the values over 4096 frames. Once the element counter is started, the appropriate data latch is enabled sending the output to the multiplier stage. The real and the imaginary parts of the output are generated in parallel by performing four real number multiplications.


Initialize all arrays and counters
Start / count frame counter (k)
Increment load counter (l_i)
Load ith component of vector in **Y(i)** latch
Load scalar in **X** latch of the ith processor
No
$l_i \geq 7$
Yes
Start / count element counter (i)
Enable address latch for reading
Two simultaneous multiplications
Two simultaneous multiplications
Retrieve stored value from memory using i as address. Disable address latch
Subtract for real part
Add for imaginary part
Shift right 6 bits
Shift right 6 bits
Accumulate
Accumulate
1
2
3

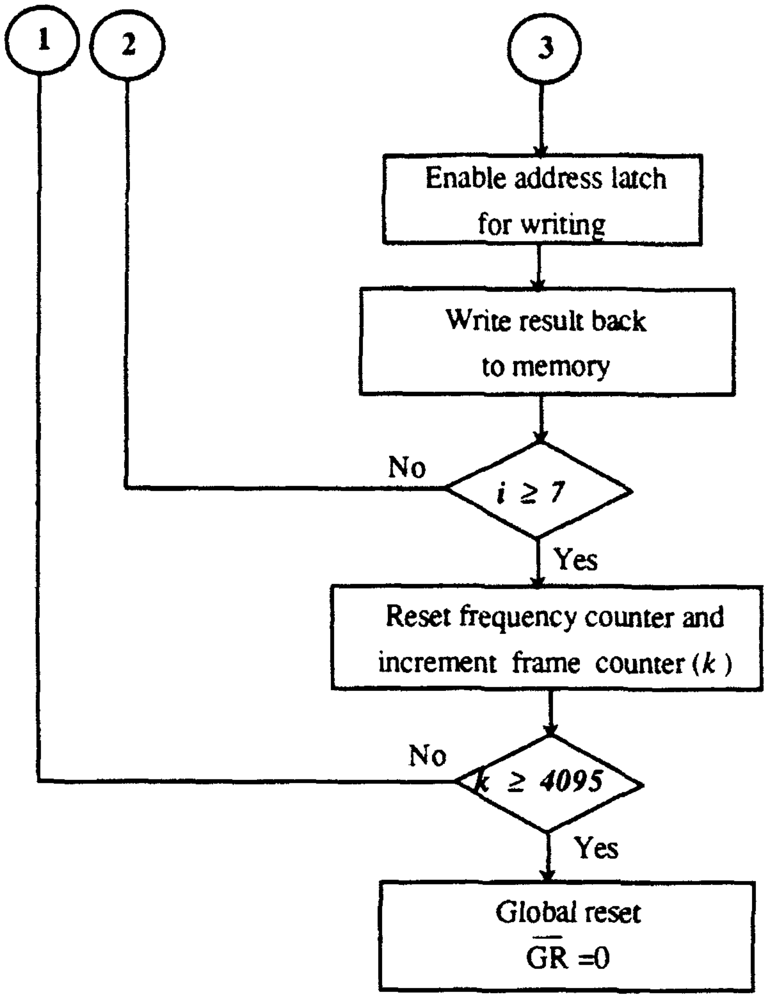


Figure 5.2 : Flowchart of operations performed to compute covariance matrix for the narrowband MUSIC algorithm.

This is done by the four eight bit multipliers in the arithmetic unit. To obtain the imaginary part, one product is subtracted from the other. Similarly the real part is obtained by adding the corresponding products. A memory read operation is performed in parallel which will read the previously computed result and is added to the newly computed element.

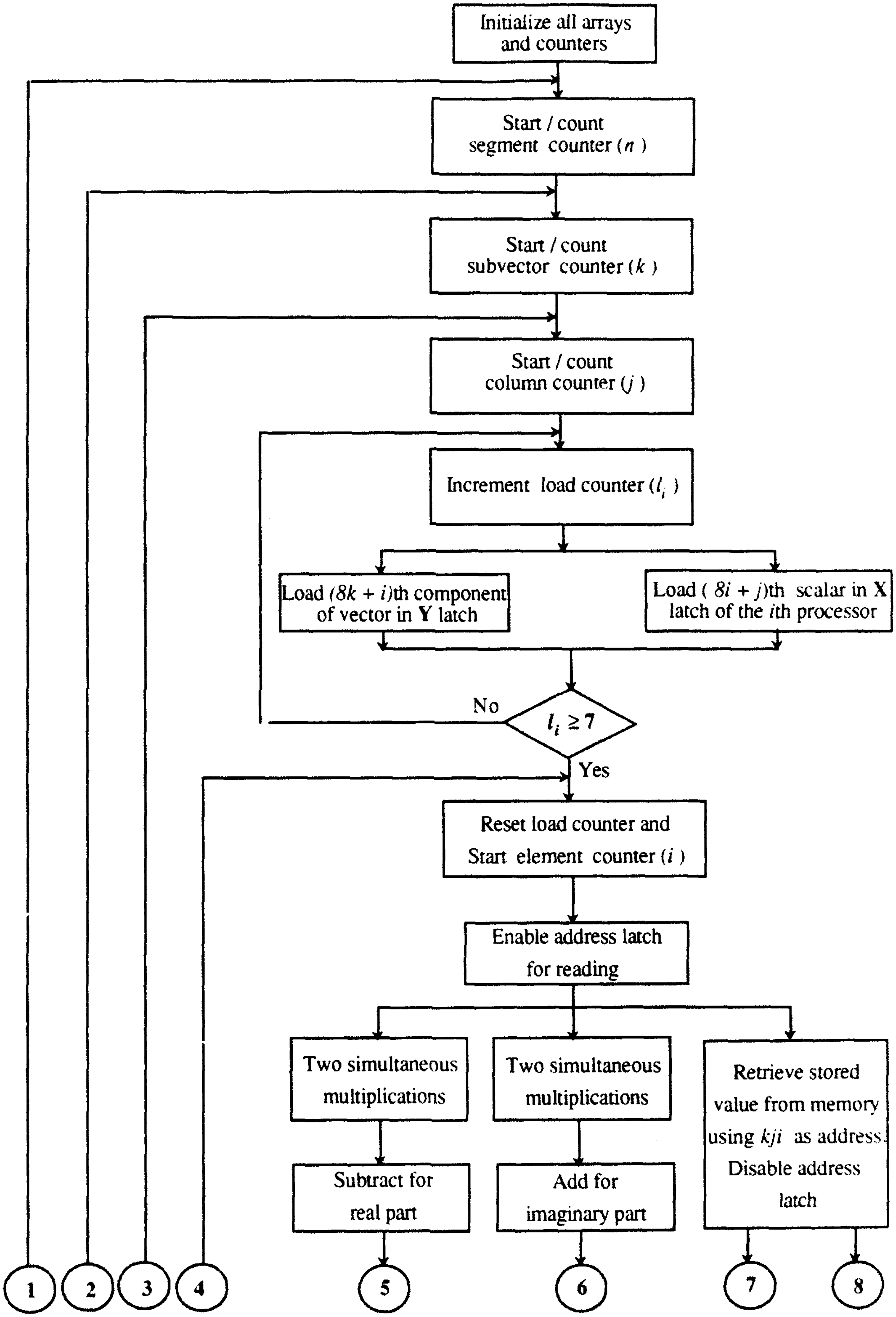
Overall 4096 such frames are accumulated. The sensor output is quantized into 8 bit real and imaginary parts and hence the word size becomes 16 bits after the multiplier stage. After the final accumulation the data becomes 28 (16+12) bits. This increases the chip area, data bus width and the memory requirements. To alleviate this problem the result is pre-shifted before accumulation by 6 bits.

Once the accumulation is complete the address latch is enabled again and the result is written back to the memory. This loop is performed 8 times as shown in the flowchart. Then the frame counter is incremented and the operations are performed 4096 times. Finally the global reset is enabled which resets all counters and memory arrays.

5.3 COVARIANCE MATRIX COMPUTATION FOR BROADBAND BASS-ALE ALGORITHM

The BASS-ALE method is a broadband algorithm which uses the eigenstructure of a temporal covariance matrix and broadband source models to estimate the DOA. Like the MUSIC algorithm the input vectors to the covariance stage are samples in the time domain. However for the BASS-ALE method operating with a system of eight sensors the input vector is a time delayed set of 64 samples. Eight samples are obtained from each sensor taken after a speccific time delay. They are then stored in a delay array before the covariance processor stage, which gives a 64 element vector. The multiplication of a 64 element with its Hermetian yields a 64x64 matrix. A parallel and pipelined architecture for this procedure will consist of an array of 64 processors with each one computing one column of the resultant matrix. A new scheme has been proposed [18] which allows the computation of the covariance matrix using an array of eight processors. An eight processor architecture is adopted as it is similar to the one proposed for the narrowband MUSIC and bilinear transformation algorithms.

The flowchart for the computation of the covariance matrix is shown in Figure 5.3. The arithmetic operations are exactly similar to the ones explained above. The major difference lies in the controlling of the number of loops and the loading of the input latches. As before the processor has 8 latches for the Y input which will recieve the broadcast vector. The X input is distinctive and is given only to one specific PE. As shown in the flowchart the control unit performs four nested loops for the BASS-ALE algorithm. The 64X64 matrix is split into 8 sub matrices each of which is 64X8 in dimension with the *i*th processor computing the *i*th submatrix. For example the 4th processor will compute the 4th submatrix which consists of the columns 25-32 in the covariance matrix. To simplify the control unit these submatrices are split up into eight 8x8 micromatrices. For each new column of the micromatrix the data has to be loaded into the PE. As before this is handled by an external load counter. The $(8k+i)$ th component of the vector is


Initialize all arrays and counters
Start / count segment counter (n)
Start / count subvector counter (k)
Start / count column counter (j)
Increment load counter (l_i)
Load $(8k + i)$th component of vector in **Y** latch
Load $(8i + j)$th scalar in **X** latch of the ith processor
No
$l_i \geq 7$
Yes
Reset load counter and Start element counter (i)
Enable address latch for reading
Two simultaneous multiplications
Two simultaneous multiplications
Retrieve stored value from memory using kji as address. Disable address latch
Subtract for real part
Add for imaginary part
1
2
3
4
5
6
7
8

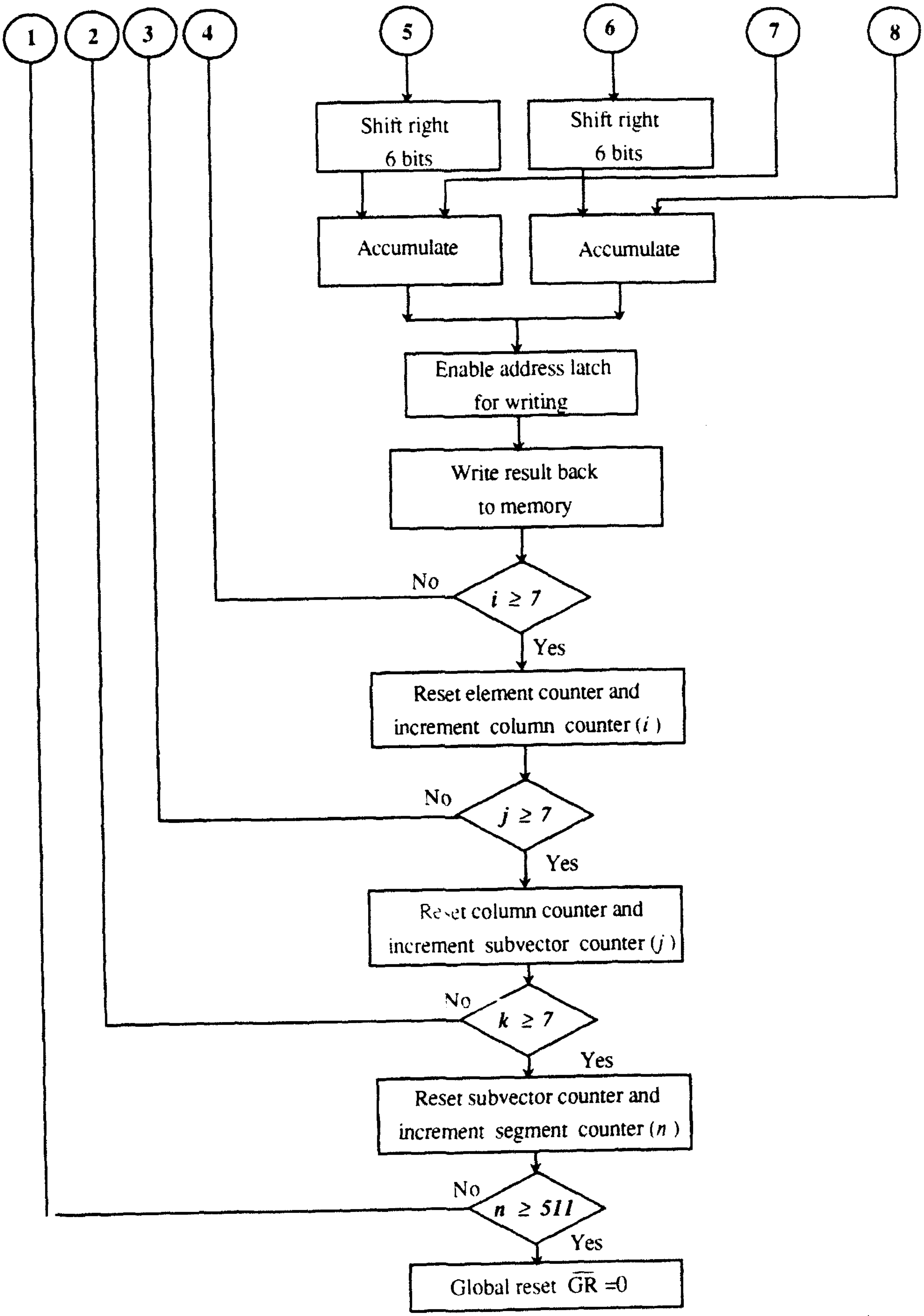
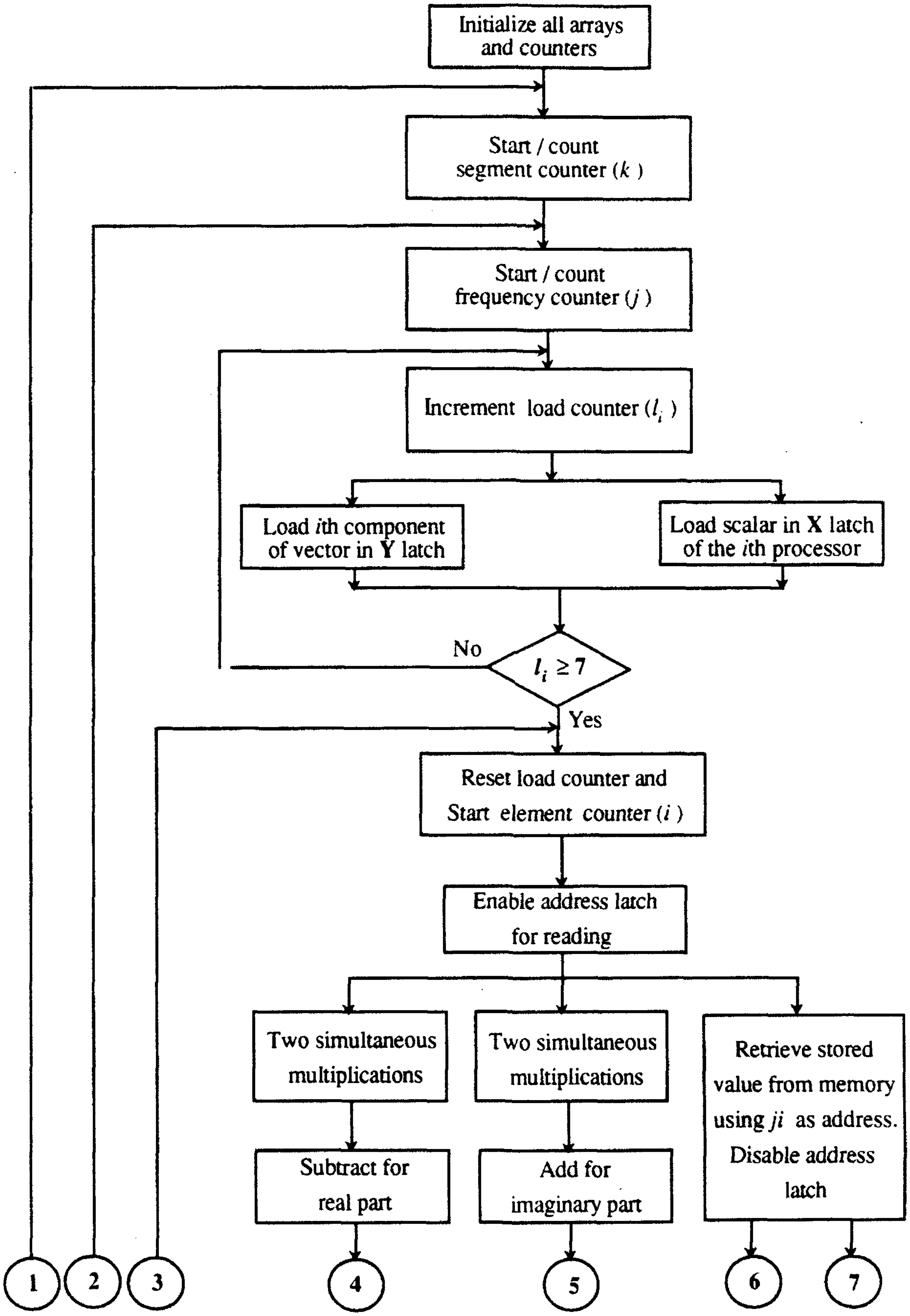


Figure 5.3 : Flowchart of operations performed to compute covariance matrix for the BASS ALE method.

loaded to all the Y latches and the *(8i+j)*th scalar for that particular submatrix is loaded in the X latch. The arithmetic operations are the same as before with four multiplications, an addition and a subtraction. Simultaneously, the word is read in from the memory using *kji* of the counters as the address. The accumulating operation is then carried out and the result written back to the memory. Once the 8 elements of the column are calculated the processor computes the rest of the micromatrix and then each segment to finish one iteration of computations. The matrix is then accumulated over 512 loops and finally averaged, the global reset signal is enabled and the matrix is passed on for the computation of eigenvectors. The calculation of the covariance matrix for the BASS-ALE algorithms involves 64 times the number of computational operations when compared to the previous case. Hence to complete one full iteration the processor takes more time and to match the processor speed with the sensor speed a delay buffer before the processor stage is suggested.

5.4 COVARIANCE MATRIX MULTIPLICATION FOR BILINEAR TRANSFORMATION ALGORITHM

The computation of the covariance matrix for the bilinear transformation method has been explained in the previous chapters. The processor outlined previously has a completely parallel and pipelined architecture but has the disadvantage of requiring a larger area and hence is not very suitable for single chip implementation. To reduce the chip size the number of complex multiplying units is reduced to one and another counter is added in the control unit. This element


Initialize all arrays and counters
Start / count segment counter (k)
Start / count frequency counter (j)
Increment load counter (l_i)
Load ith component of vector in **Y** latch
Load scalar in **X** latch of the ith processor
No
$l_i \geq 7$
Yes
Reset load counter and Start element counter (i)
Enable address latch for reading
Two simultaneous multiplications
Two simultaneous multiplications
Retrieve stored value from memory using ji as address. Disable address latch
Subtract for real part
Add for imaginary part
1
2
3
4
5
6
7

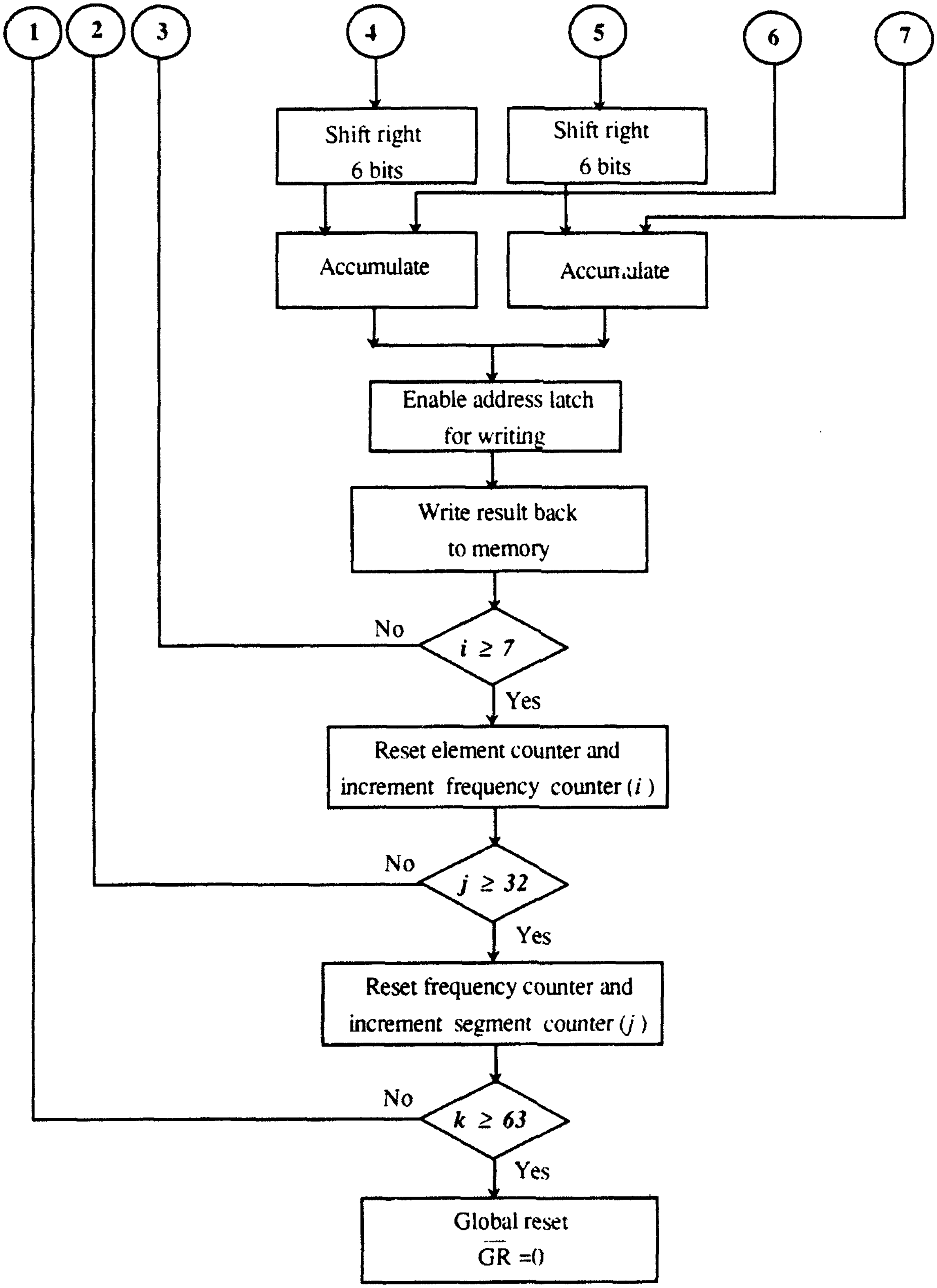


Figure 5.4 : Flowchart of operations performed to compute covariance matrix for the bilinear transform method.

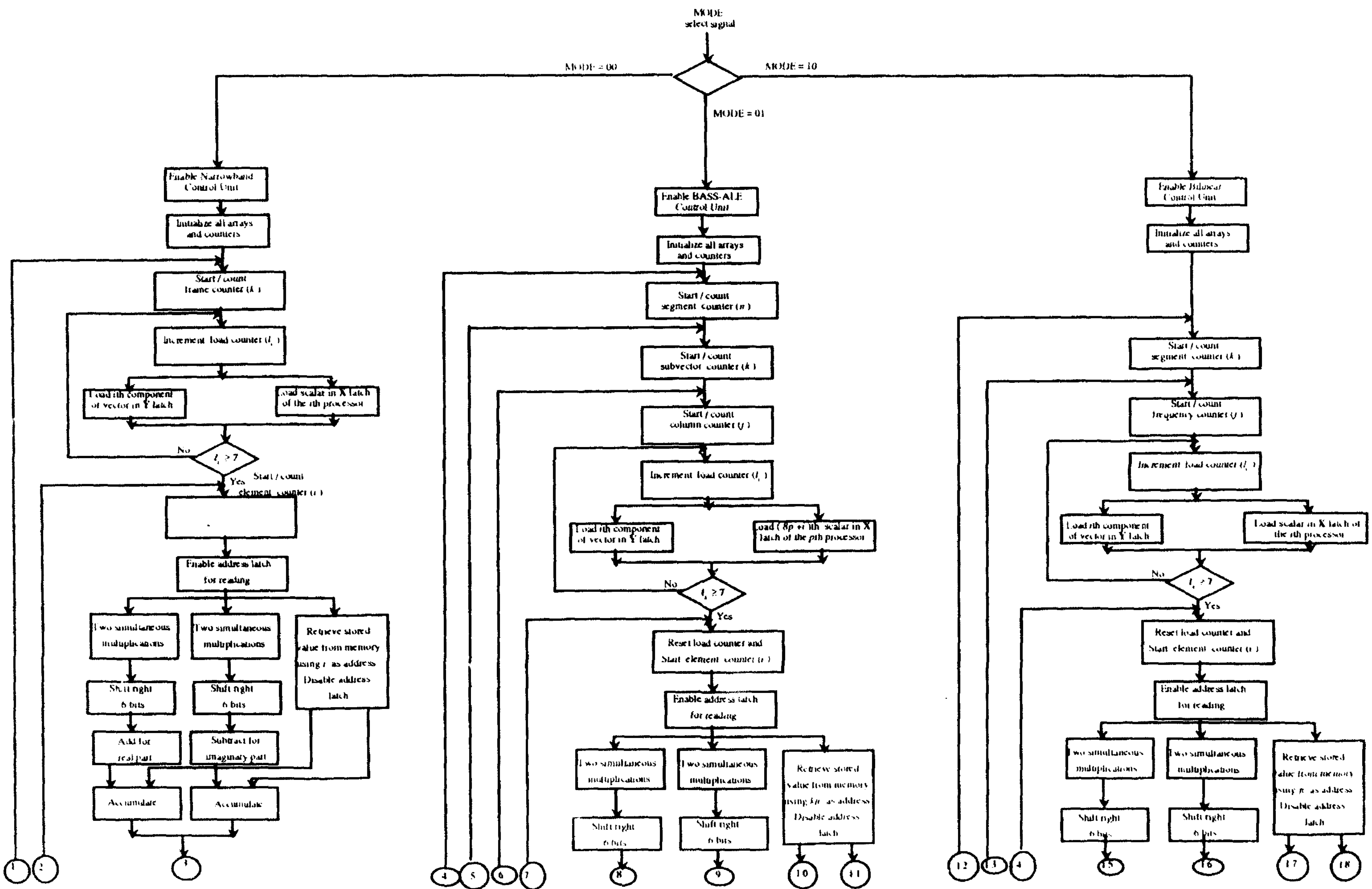
counter controls the computation of the individual elements of the column which are now computed sequentially instead of parallelly. This configuration can be easily mapped onto a genralized architecture for covariance matrix computation. The flow chart for the bilinear transformation operation is shown in Figure 5.4. The computation of the covariance matrix for this algorithm is done in the frequency domain over a range of 33 frequencies. One covariance matrix is generated at each frequency bin and then averaged over 64 frames. The arithmetic operations are simlilar to previously described operations, but in this case as the averaging is done over only 64 frames the initial preshift by 6 bits is enough, and the shifting out after accumulation is not required.

A combined flowchart for the computation of the covariance matrix for all three algorithms is shown in Figure 5.5. A two bit mode select signal is used to select the desired algorithm. The control is then transferred to the individual control units which are driven by the system clock. The system can be reset at any time by pulling up the global reset (GR) signal which is usually generated by the control units after the completion of one frame of operations.

5.5 PROCESSOR ARCHITECTURE

A block diagram of the combined covariance matrix processor is shown in Figure 5.6. The architecture basically consists of three parts:

1. The input loading stage
2. The arithmetic unit
3. The control units



MODE select signal
MODE = 00
MODE = 01
MODE = 10
Enable Narrowband Control Unit
Initialize all arrays and counters
Start / count frame counter (k)
Increment load counter (l_c)
Load ith component of vector in Y latch
Load scalar in X latch of the ith processor
$l_c \geq 7$
No
Yes
Start / count element counter (i)
Enable address latch for reading
Two simultaneous multiplications
Shift right 6 bits
Add for real part
Subtract for imaginary part
Retrieve stored value from memory using i as address Disable address latch
Accumulate
Accumulate
Enable BASS-ALE Control Unit
Initialize all arrays and counters
Start / count segment counter (n)
Start / count subvector counter (k)
Start / count column counter (j)
Increment load counter (l_c)
Load ith component of vector in Y latch
Load ($8p + i$)th scalar in X latch of the pth processor
$l_c \geq 7$
Reset load counter and Start element counter (i)
Enable address latch for reading
Two simultaneous multiplications
Shift right 6 bits
Retrieve stored value from memory using kn as address Disable address latch
Enable Bilinear Control Unit
Initialize all arrays and counters
Start / count segment counter (k)
Start / count frequency counter (j)
Increment load counter (l_c)
Load ith component of vector in Y latch
Load scalar in X latch of the ith processor
$l_c \geq 7$
Reset load counter and Start element counter (i)
Enable address latch for reading
Two simultaneous multiplications
Shift right 6 bits
Retrieve stored value from memory using n as address Disable address latch
1
2
3
4
5
6
7
8
9
10
11
12
13
4
15
16
17
18

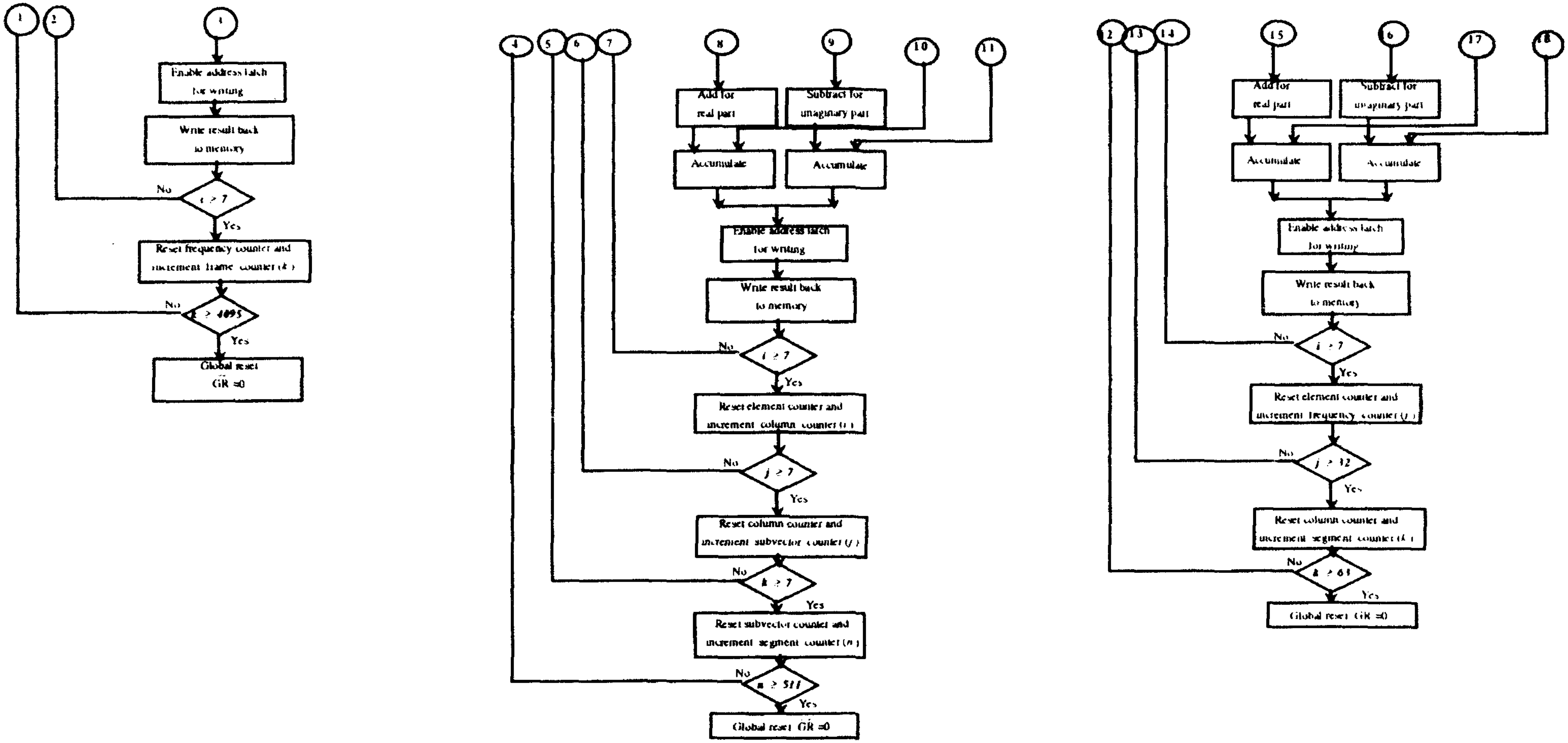


Figure 5.5 :- Flowchart for the operations performed by the combined covariance matrix processor

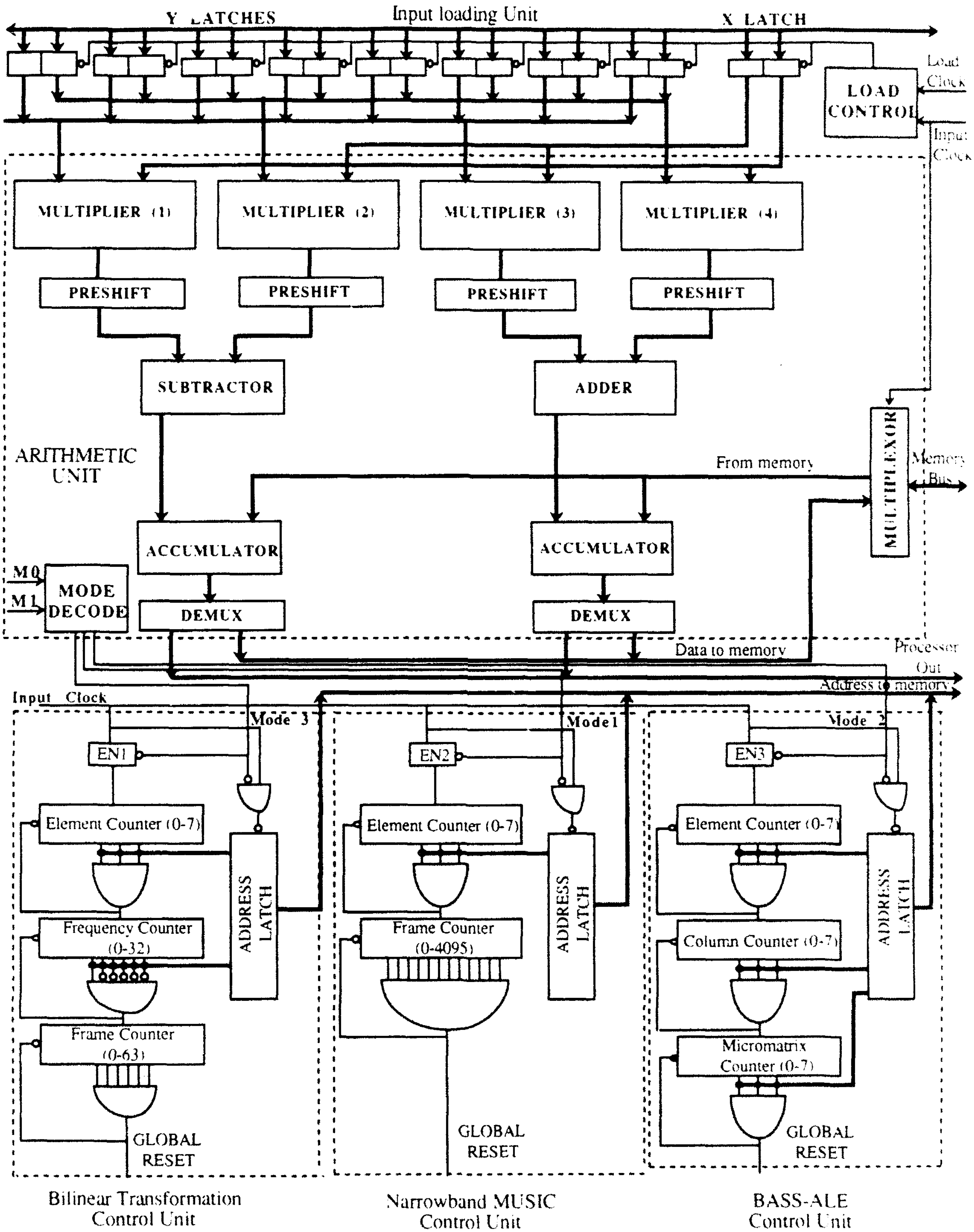


Figure 5.6: Block diagram of combined covariance matrix processor

Figure 5.7 shows the mode select unit that is used to select the desired algorithm. The input loading stage consists of eight input latches for the **Y** vector and one for the **X** scalar and a load control unit which latches on the data at the appropriate clock pulse. Figure 5.8 shows the load control unit. The unit has two three bit counters and two 3 to 8 decoders. The latch counter is used to count the clock pulses and the decoder selects the appropriate latch according to the clock. One counter is used to latch the input data and is driven by the load clock. As shown in the flowcharts the load clock is received when the loading operation takes place. Once the data for one particular cycle is latched in, the load clock is disabled and the input clock which synchronizes the arithmetic operations is enabled. The input clock is used to drive the enable counter and the enable decoder which enables the appropriate buffer. The data from the latch is then placed on the internal data bus which is connected to the multipliers. As all the latches are connected to the same internal bus, a tristate buffer is used after the latch to prevent data corruption. The arithmetic unit has four multipliers, an adder, a subtractor and two accumulators. The control unit has three separate control modules for three algorithms. The functions and operation of these units are discussed in detail in the next chapter.

In the next section the behavioral simulation of the processor using VHDL is considered. The architecture is verified at the module level and all the architectural considerations were taken care of.

5.5.1 Powerview 5.1

Powerview 5.1, from Viewlogic is a CAD package [19] that has the capability of simulating analog/digital architectures from the logic gate level to the module level. It offers a wide variety of choices to the designer who can

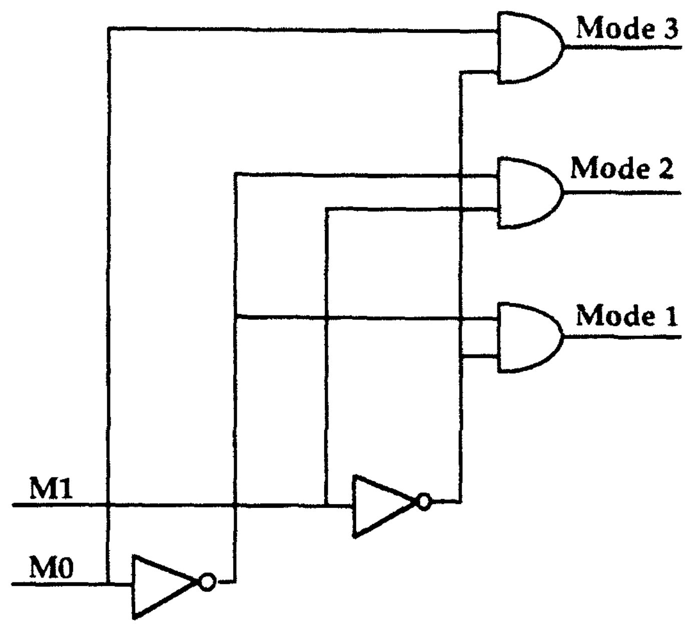


Figure 5.7 : Schematic of mode decode unit

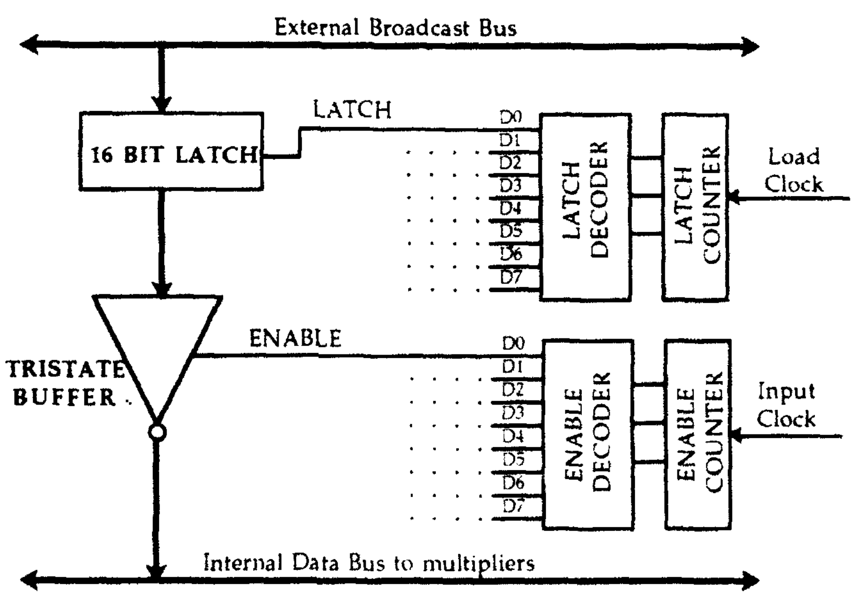


Figure 5.8 : Schematic of load control unit

choose from a standard cell library or can construct his own from the basic logic gates. The package also supports a variety of tools ranging from Hspice, PCB design, FPGA analysis and VHDL.

To perform a behavioral simulation of the proposed architecture, VHDL code was written for all the basic modules. Appendix A contains all the VHDL code for the various modules used in the architecture. The VHDL file was simulated using Viewsim, the simulation tool available on Powerview. The VHDL modules were converted into schematic symbols and called as components inside Viewdraw, Powerview's schematic editor. The modules were then connected together to form the processor model. The architecture was once again simulated using Viewsim and the results were plotted using Viewtrace.

5.5.2 Behavioral Simulation of the Architecture

The first step in the behavioral simulation was to write VHDL code for all the basic modules in the processor. The processor was then constructed from them. Figure 5.9 shows the Viewdraw schematic of the input loading block. The figure shows the latches which form the input block and the load control unit. The top set of latches in the figure is the **Y** vector latch and the separate one is the **X** latch. The load control unit is shown at the top.

Figure 5.10 shows the Viewdraw schematic of the control unit for the narrowband MUSIC algorithm. The three bit element counter is connected through an AND gate to the clock input of the twelve bit frame counter. The output of the frame counter is given to a 12 bit AND block which generates the global reset signal. The address latch is connected to outputs of the element counter. It is enabled when both the control unit and the counter is

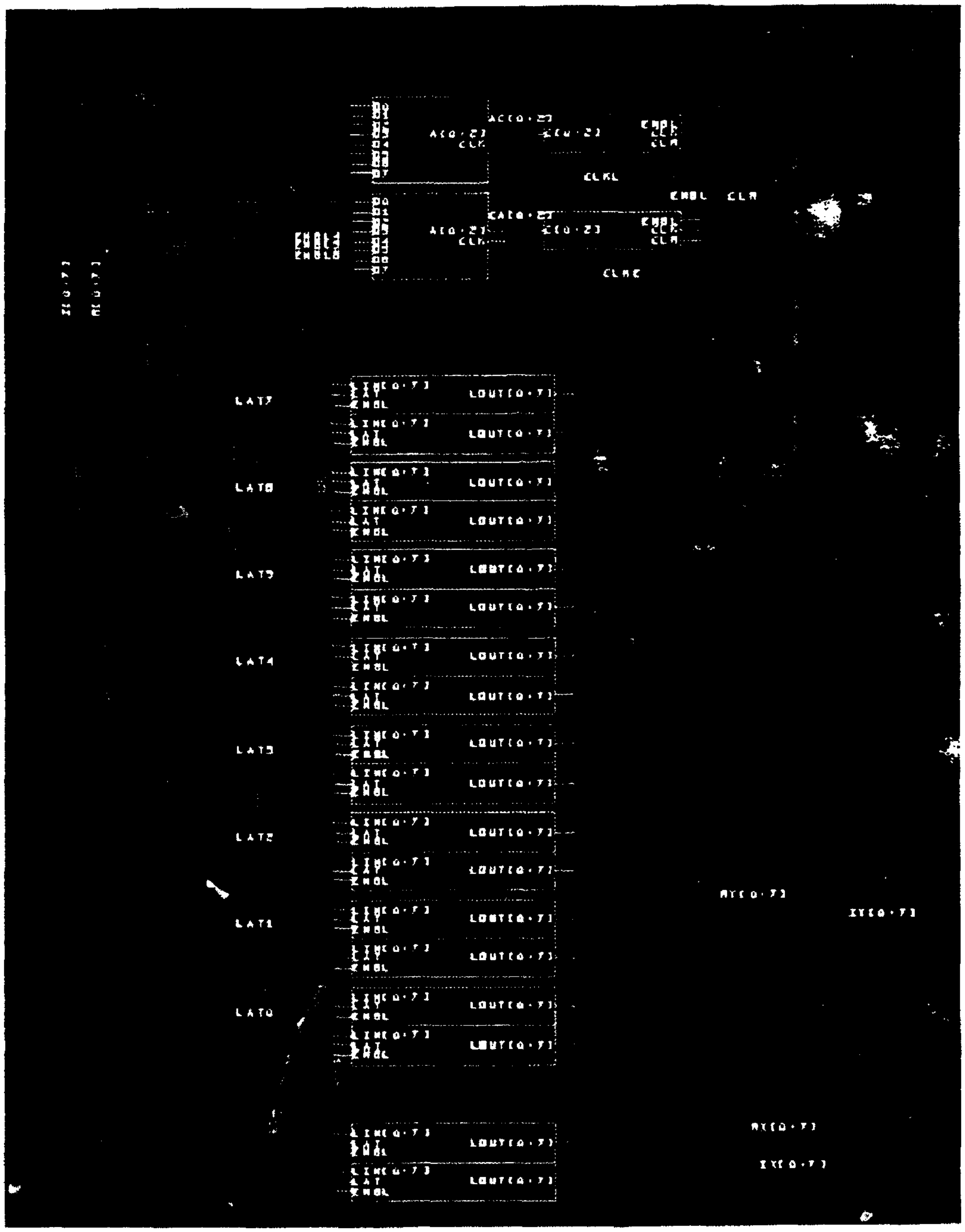


Figure 5.9 : Viewdraw Schematic of input loading block

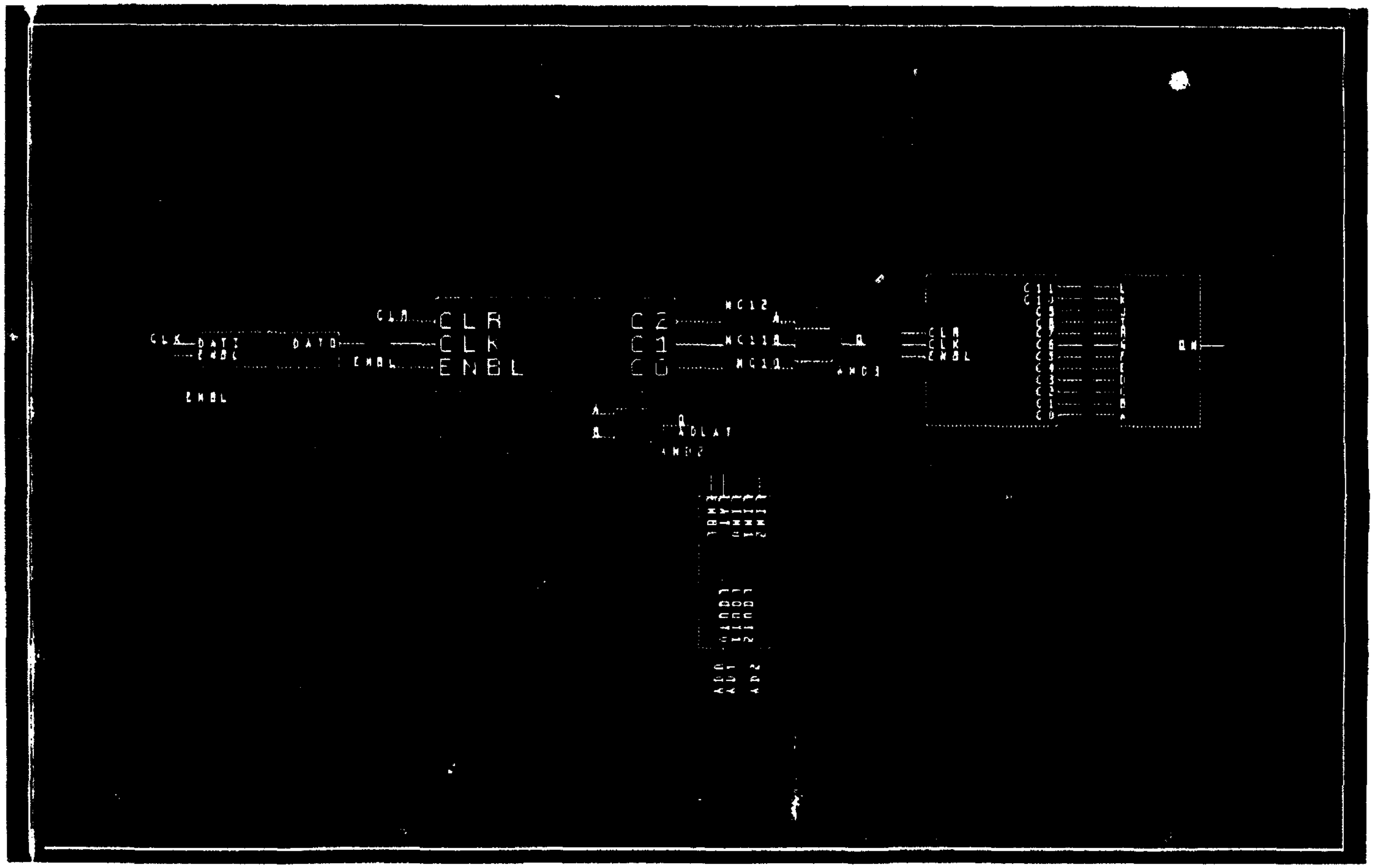


Figure 5.10 : Viewdraw schematic of Narrowband MUSIC algorithm control unit

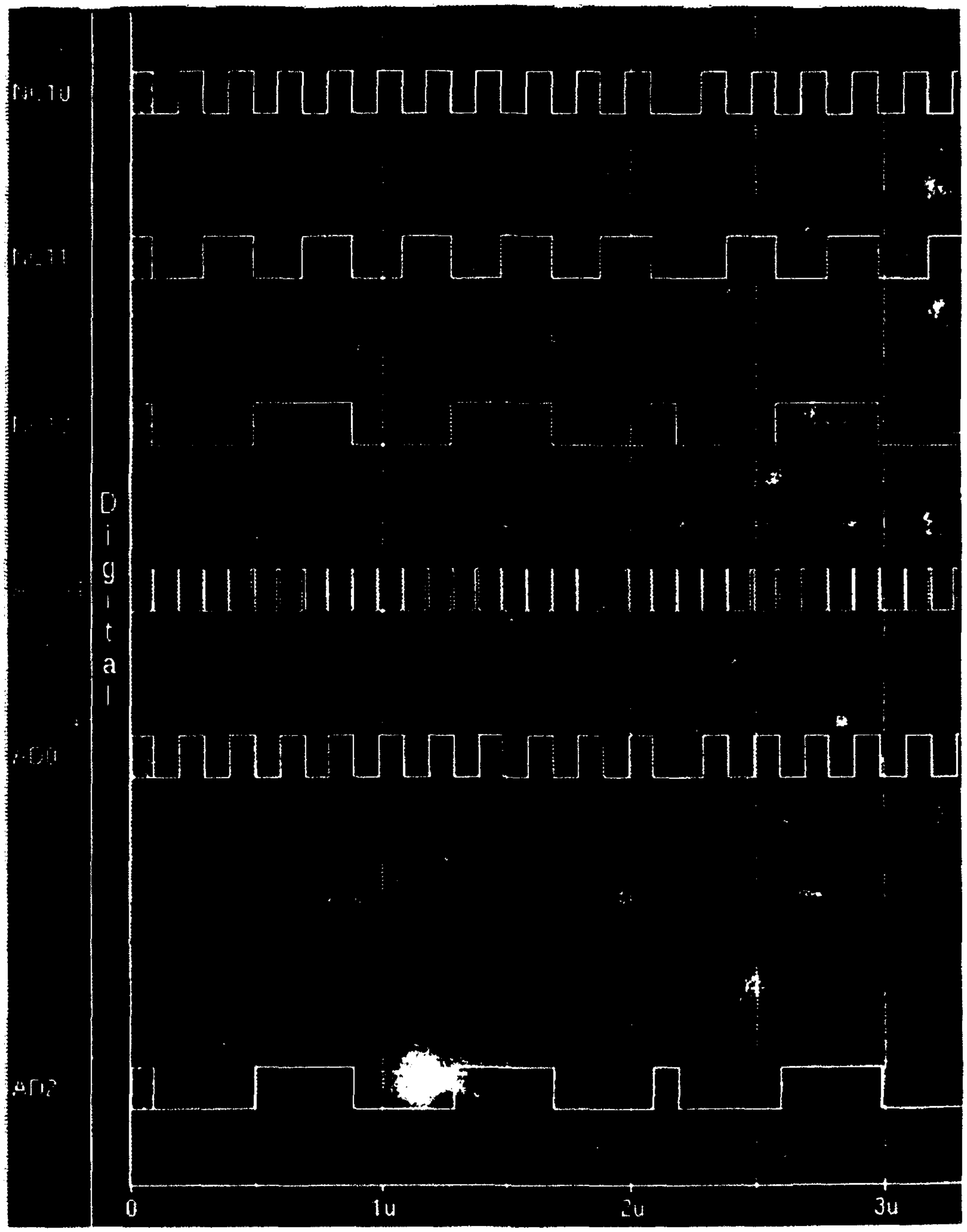


Figure 5.11 . Viewsim results for simulation of narrowband MUSIC control unit

enabled. Figure 5.11 shows the Viewsim results of the simulation of the control unit. It can be seen that the counters generate the required signals according to the clock input. NC10-NC12 are the outputs of the element counter and *ADLAT is the input clock. AD0-AD2 are the address bits that are* obtained at the output of the address latch.

Figure 5.12 shows the Viewdraw schematic of the control unit for the broadband BASS-ALE algorithm. The three bit element counter is connected through an AND gate to the clock input of the three bit column counter which in turn is similarly connected to the input of the micromatrix counter. The output of the micromatrix counter is given to a 3 input AND gate which generates the global reset signal. The outputs of all the three counters are stored in the address latch. Figure 5.13 shows the Viewsim results of the simulation of the control unit. The counter outputs are BC10-BC12 (element counter), BC20-BC22 (column counter) and BC30-BC32 (micromatrix counter). ADOUT0-ADOUT9 are the address bits that are generated by the control unit.

Figure 5.14 shows the Viewdraw schematic of the control unit for the bilinear transformation algorithm. The three bit element counter is connected through an AND gate to the clock input of the six bit frequency counter. The output of the frame counter is given to a logic block which generates a reset signal when the input bits are 100000(32). This simple logic bolck consists of a NOR gate with an inverter attached to the MSB input. The output of this logic gate is used to reset the frequency counter and acts as a clock to the six bit frame counter whose output generates the global reset signal. The outputs of the element counter and the frequency counter are connected to the address latch. Figure 5.15 shows the Viewsim results of the simulation of the control unit. The counter outputs are BC10-BC12 (element

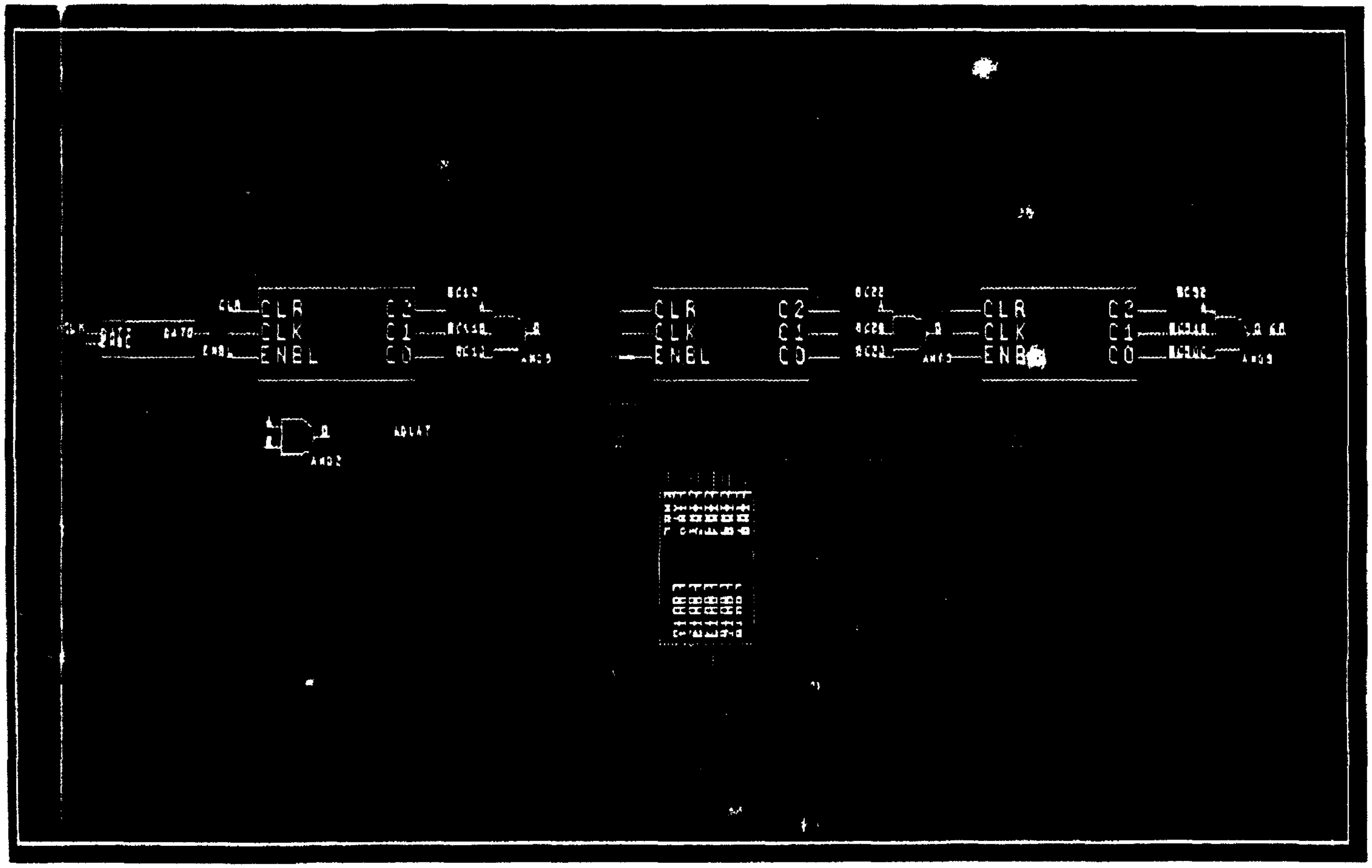


Figure 5.12 : Viewdraw schematic of control unit for BASS-ALL algorithm

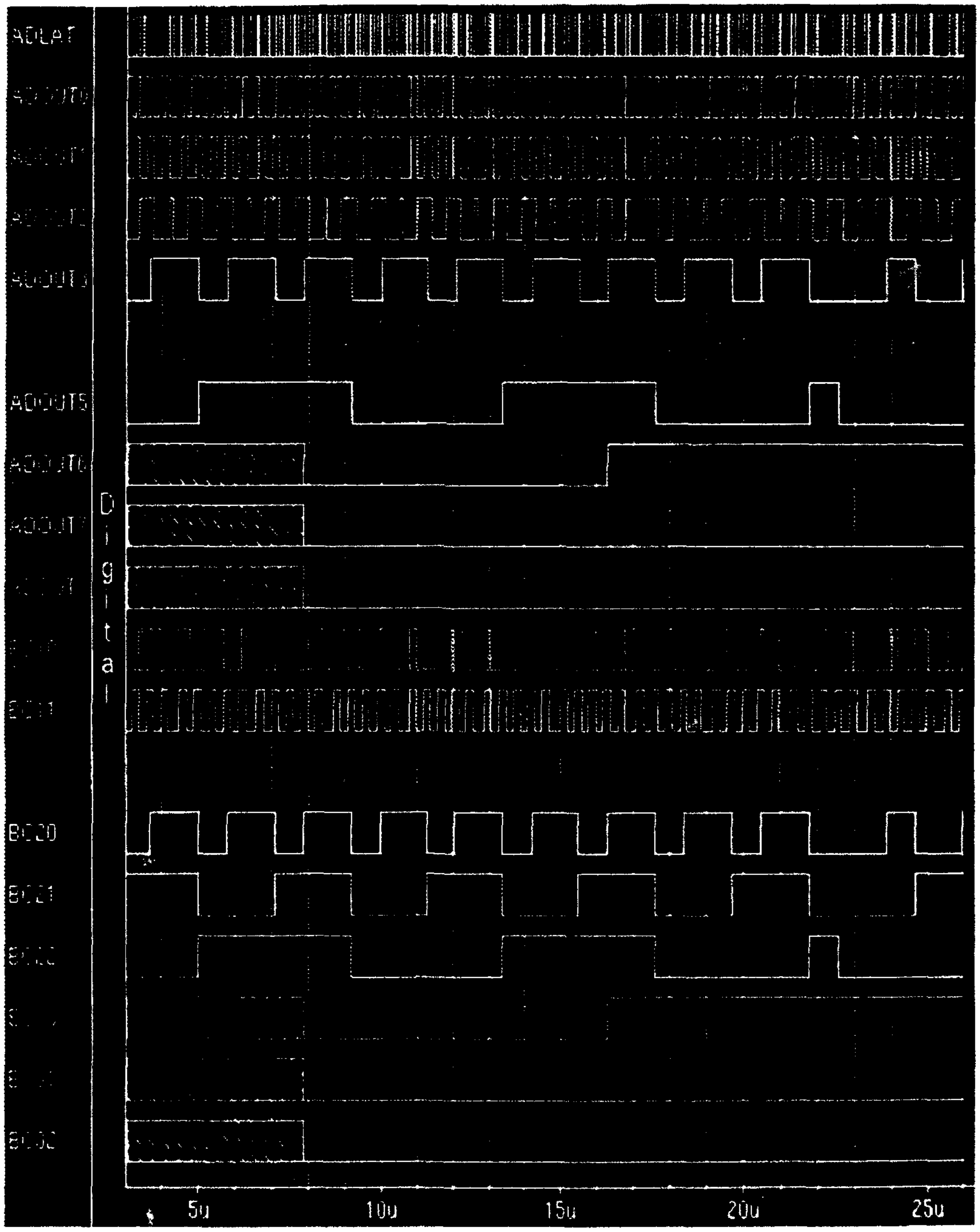


Figure 5.13 : Viewsim simulation results of BASS-ALE control unit

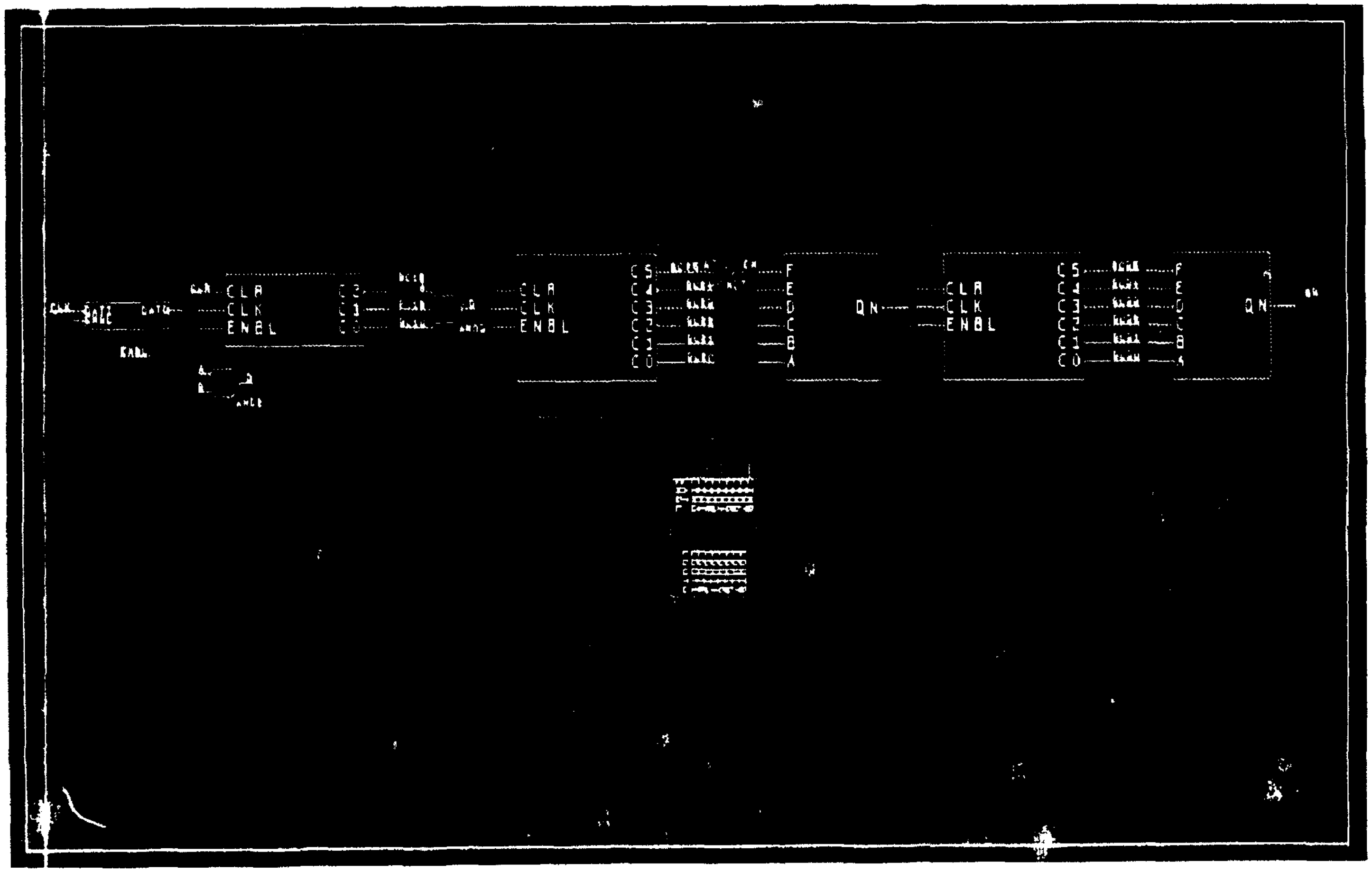


Figure 5.14 : Viewdraw schematic of Bilinear Transformation control unit

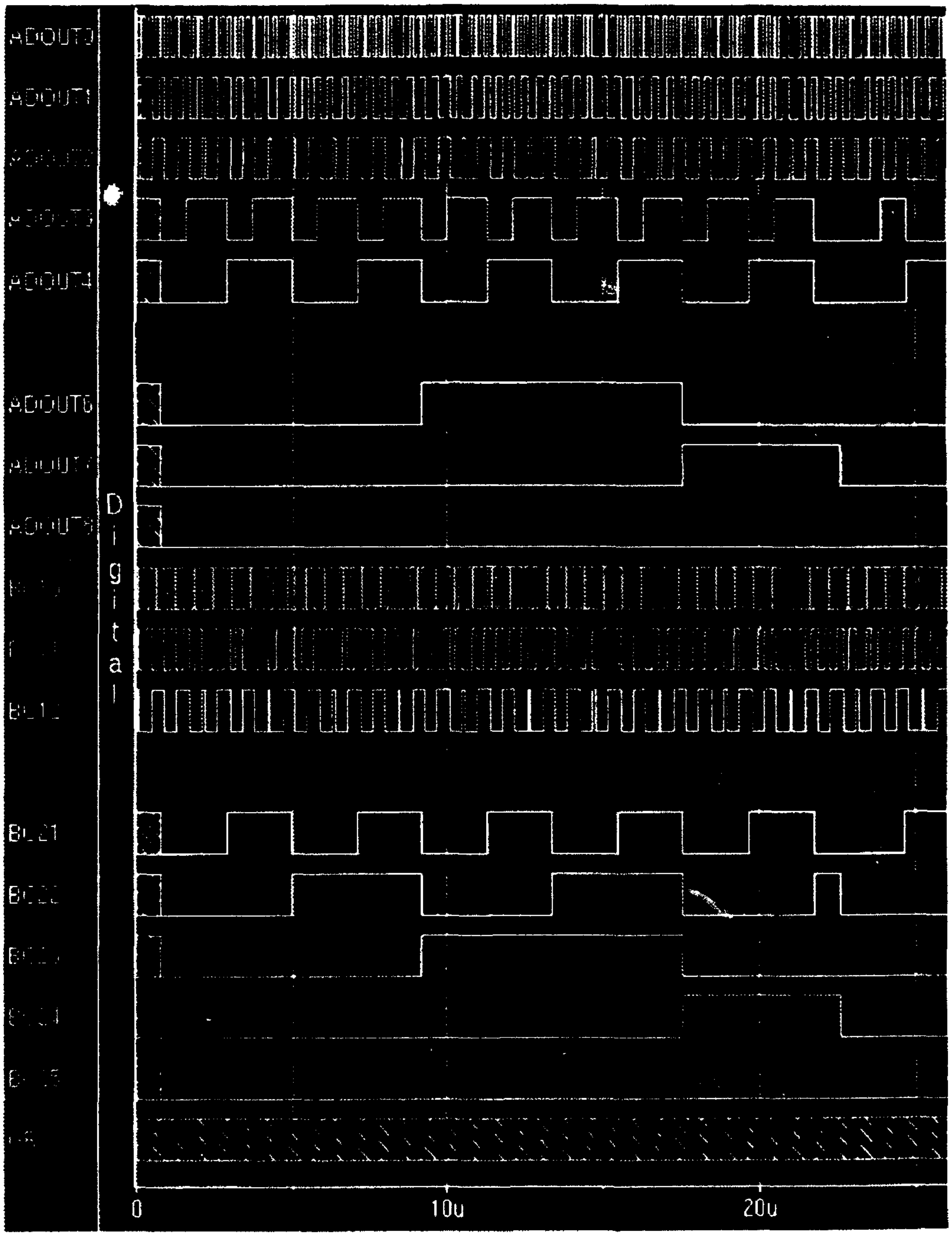


Figure 5.15 : Viewsim results of the bilinear control unit

counter) and BC20-BC25 (frequency counter). ADOUT0-ADOUT9 are address bits that are generated by the control unit.

The complete processor was then connected using Viewdraw. The schematic of the complete processor is shown in Figure 5.16. The data from the input latches is fed into tl e arithmetic unit which computes the complex number multiplication and gives the result to the accumulator. The other input of the accumulator is from the RAM. The memory result of the previous accumulation is read in using the address supplied from the address bus. Once the complete cycle of operations are complete the control unit generates the global reset signal which is used to place the output of the accumulator on the processor out pins. The Viewsim results of the processor simulation are shown in Figure 5.17. RA1, RA2, IA1, IA2 are the four inputs given to the multipliers on each clock pulse and ROUT and IOUT are the outputs of the processor. Consider the case when the inputs are 01, 02, FC and FD. The input vectors are 1 +iFC (a+ic) and 2 +iFD (b+id). The outputs of the four multipliers will be:

ab = 1 x 2 = 2

cd = FC x FD = F90C

bc = 2 x FC = 1F8

ad = 1 x FD = FD

The real and imaginary outputs ROUT and IOUT will therefore be:

real out = ab + cd = 2 + F90C = F90E

imaginary out = bc - ad = 1F8 - FD = FB

The processor architecture is hence verified.

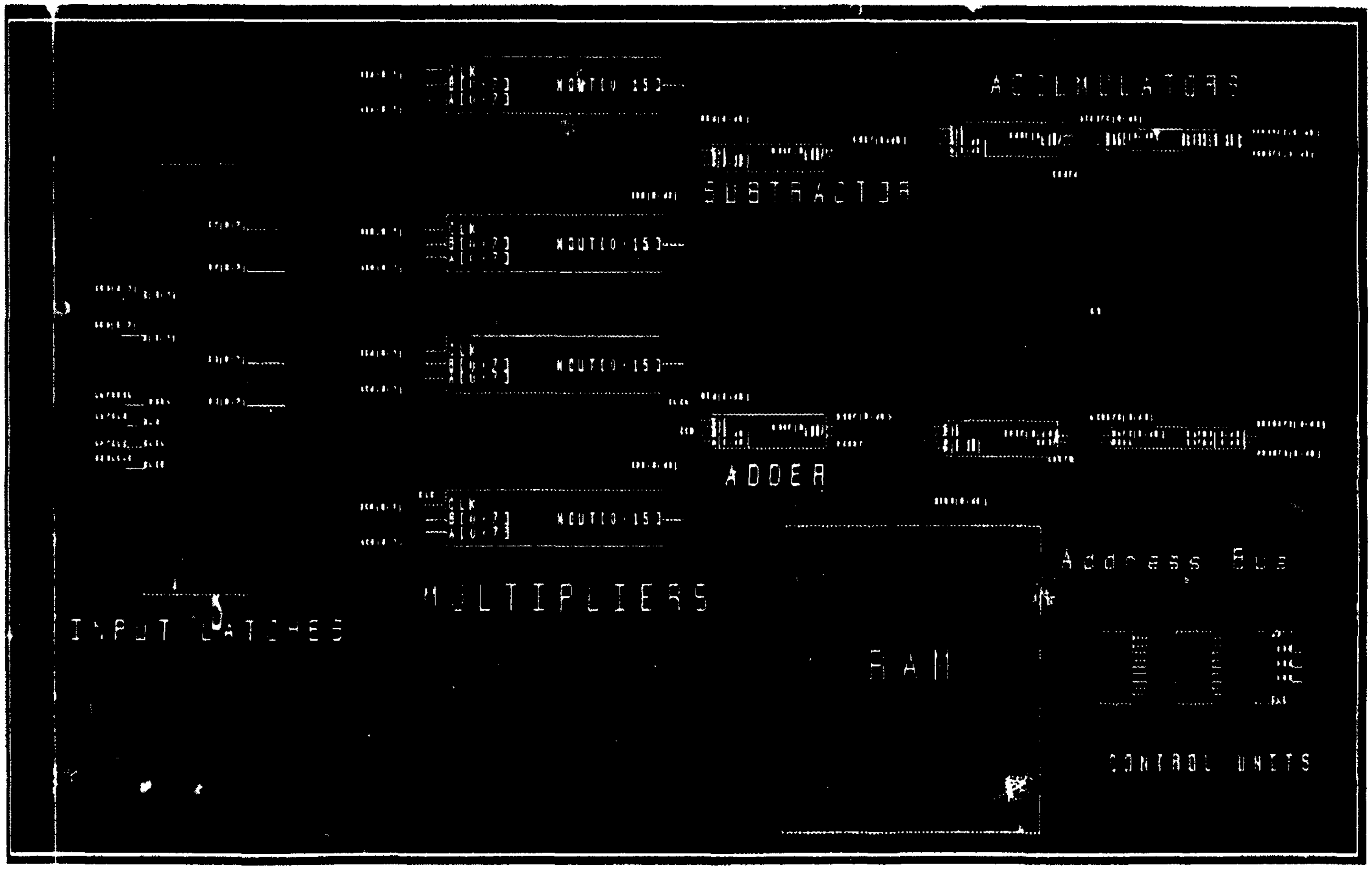


Figure 5.16 Viewdraw top level Schematic of Combined covariance matrix processor

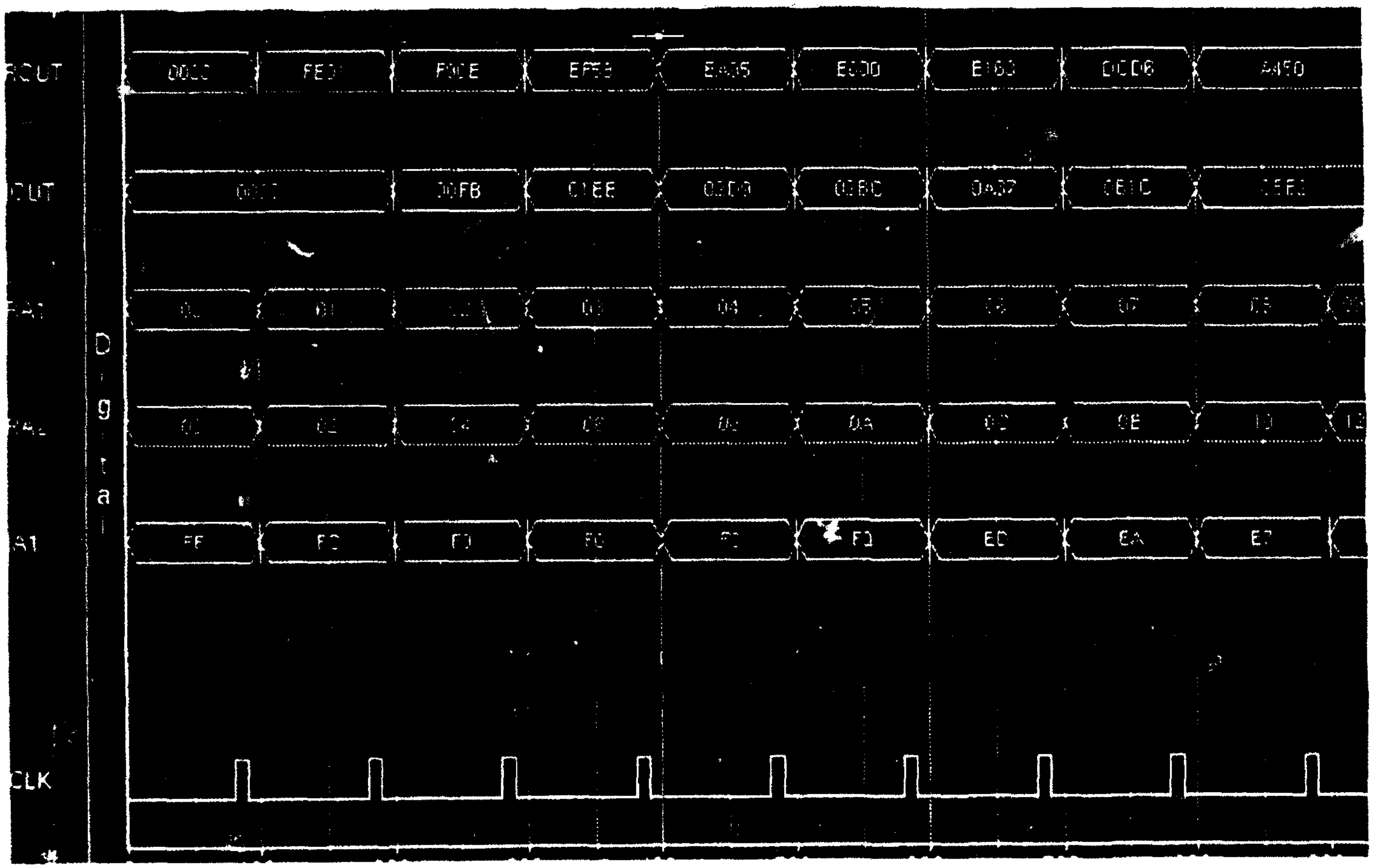


Figure 5.17 Viewsim results for simulation of covariance matrix processor

CHAPTER 6

VLSI Implementation

6.1 INTRODUCTION

The VLSI implementation of the processor described in the previous chapter involves a detailed design of the individual modules and transistor level optimization to provide a chip which can perform the required operations in the required time frame. During the VLSI design various considerations such as the selection of multiplier and adder architectures and number representation were taken into account, and the chip design was carried out accordingly.

The VLSI *simulation and implementation* was carried out using Mentor Graphics Generator Development Tools 5.3. The GDT tools were used to perform the transistor and logic level simulation on the chip and conduct a timing analysis. The layout of the chip was generated and verified using the AutoCells feature in GDT.

6.2 GENERATOR DEVELOPMENT TOOLS

The implementation and simulation of the combined covariance matrix processor has been done using Mentor Graphics GDT on the Sun Sparcstations. In this section the various GDT tools used to simulate and lay out the ASIC are described.

6.2.1 GDT Lxcells - generation of basic gates

The first objective in constructing and simulating the processor on GDT is to generate basic cells and their layouts. This is done by using the Lxcells Utility [20]. Lxcells provides a Cell Data File (CDF) which is a flexible database that contains a cell technology library, default values for generators and cell descriptions. This information is used by the cell generators in the Lxcells to generate the behavioral models and layouts of the basic cells. The technology used for this particular process is the 0.8μ CMOS technology available through MOSIS.

The basic gates were first generated using cell generators available in Lxcells. The transistor sizes were optimized and the layout was created for the gates. A netlist for the cells was generated and icons were defined so that the cells could be used in Led.

6.2.2 GDT Led - Schematic creation

Led is the graphics editor available on GDT which supports layout and schematic creation[21]. It was used to create the schematic of the processor inside GDT. The basic gates were used to form bigger modules such as flip flops, latches and full adders which were then used to form the larger arithmetic and control units. Netlists for various modules were created and simulated using Lsim.

6.2.3 GDT Lsim - Simulations

Lsim is a mixed-signal multi-level simulation tool available on GDT. This means that Lsim has the capability to incorporate M language and netlist descriptions at any hierarchical level. It also allows the user to simulate the model using switch, logic and adept modes on different parts simultaneously. It also provides extensive debugging tools to help in error checking and correction.

The various modules were simulated using both the switch and adept modes in Lsim. The switch mode gives the switching level simulation of all gates in the circuit and can be used initially to verify the accuracy of the circuit that has been created. The input to Lsim is the netlist file that is created from the schematic modules inside Led. The modules were then simulated in the adept mode which gives a more detailed timing analysis of all the transistor inside the modules. The adept simulation gives idea of the speed of the circuit which was then optimized to fit the timing requirements.

6.2.4 GDT AutoCells - layout generation, compaction and routing

AutoCells is an automatic routing tool for laying out circuits. It can perform fully automatic and interactive layout and control the aspect ratio of the layout to fit the block into the chip's floorplan. The input to Autocells consists of a netlist, the basic cell blocks and the control parameters. The basic cells were generated by the Lxcells layout generators for the basic gates that were created to be used in the schematic.

6.3 PROCESSOR IMPLEMENTATION

A Led schematic of the combined covariance processor is shown in Figure 6.1 which corresponds to the block diagram shown in Figure 5.6. As described in Section 5.5 the processor can be basically separated into three functional parts. The detailed design and operation of these three parts are described below.

6.3.1 The input loading stage.

The input stage consists of 9 sixteen bit latches and a load control unit. A Led schematic of the input stage is shown in Figure 6.2. Eight of the input latches are used to hold the **Y** vector and the ninth one is loaded with the **X** scalar. The sixteen bit latch as shown in Figure 6.3 contains 8 bits for the real part and 8 bits for the imaginary part. The load control unit is shown in Figure 6.4. It can either be outside the chip in which case it will drive the input stages of all eight processors in the architecture or it can be placed inside the processor and driven by an external clock. In the implementation of this processor the load control unit has been placed inside the cell. The load control unit consists of two three bit counters which provide the latch address and two decoders which interpret the address and enable the appropriate latch signal. The Led schematic of the 3x8 decoder used in the control unit is shown in Figure 6.5. The load control provides 2 control signals. One is the latch control signal which dictates which latch is to be loaded at the particular time from the external broadcast bus. The other is the enable control which provides the signal to place the latch contents onto the processor data

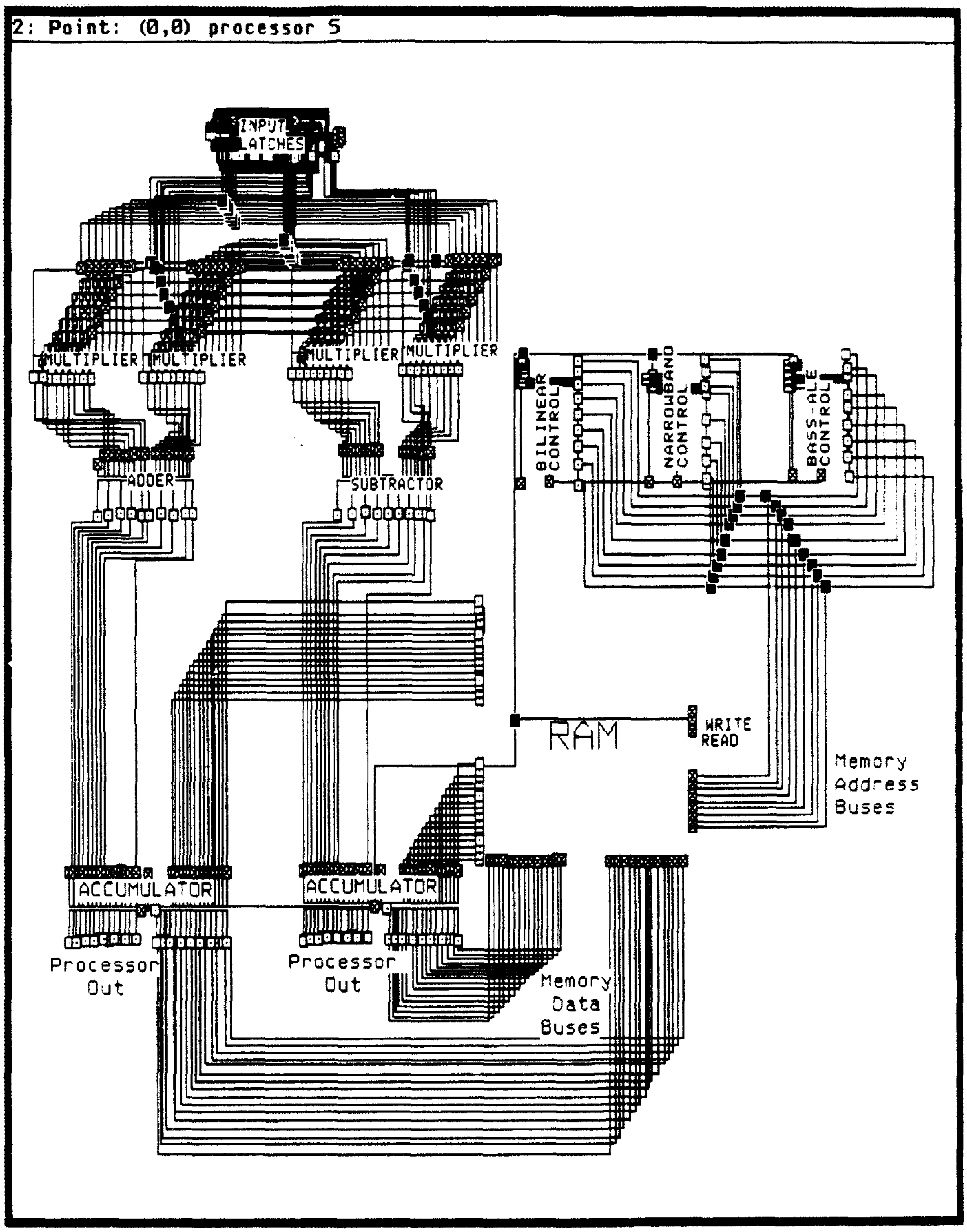


Figure 6.1 : Led Schematic of the combined covariance matrix processor

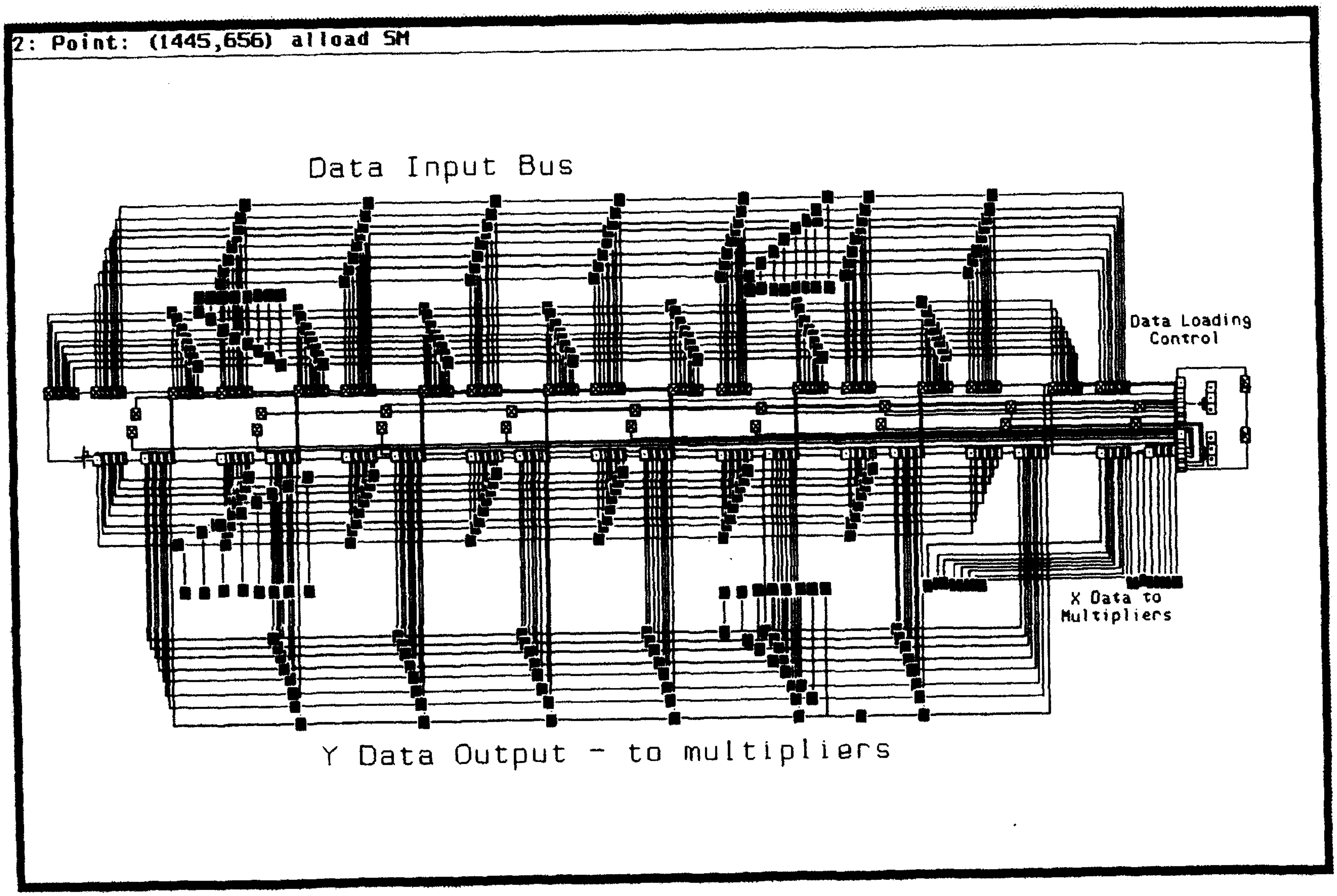


Figure 6.2 : Led Schematic of input loading stage

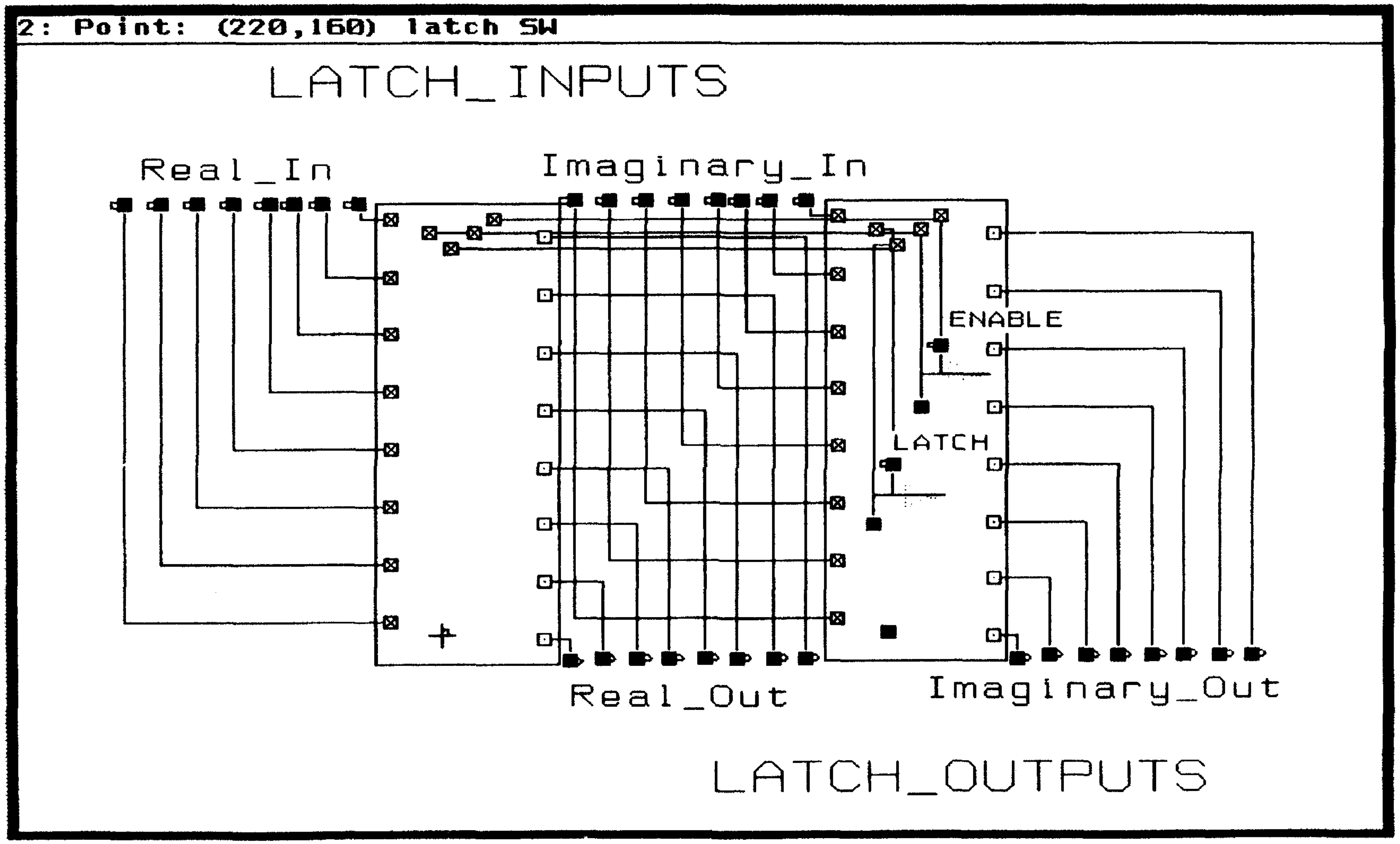


Figure 6.3 : Schematic of 16 bit latch used in the input stage

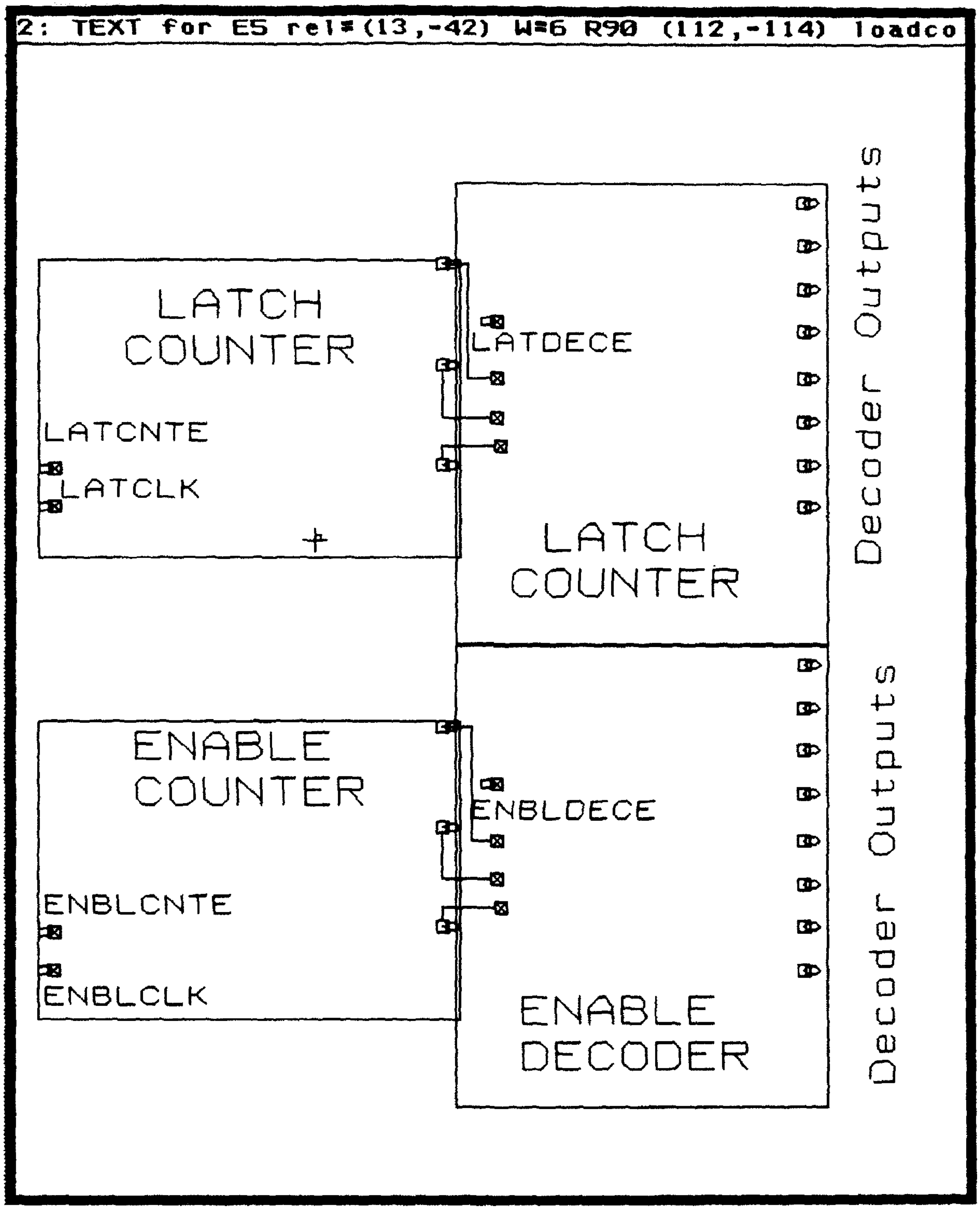


Figure 6.4 : Led Schematic of control unit for input loading operation

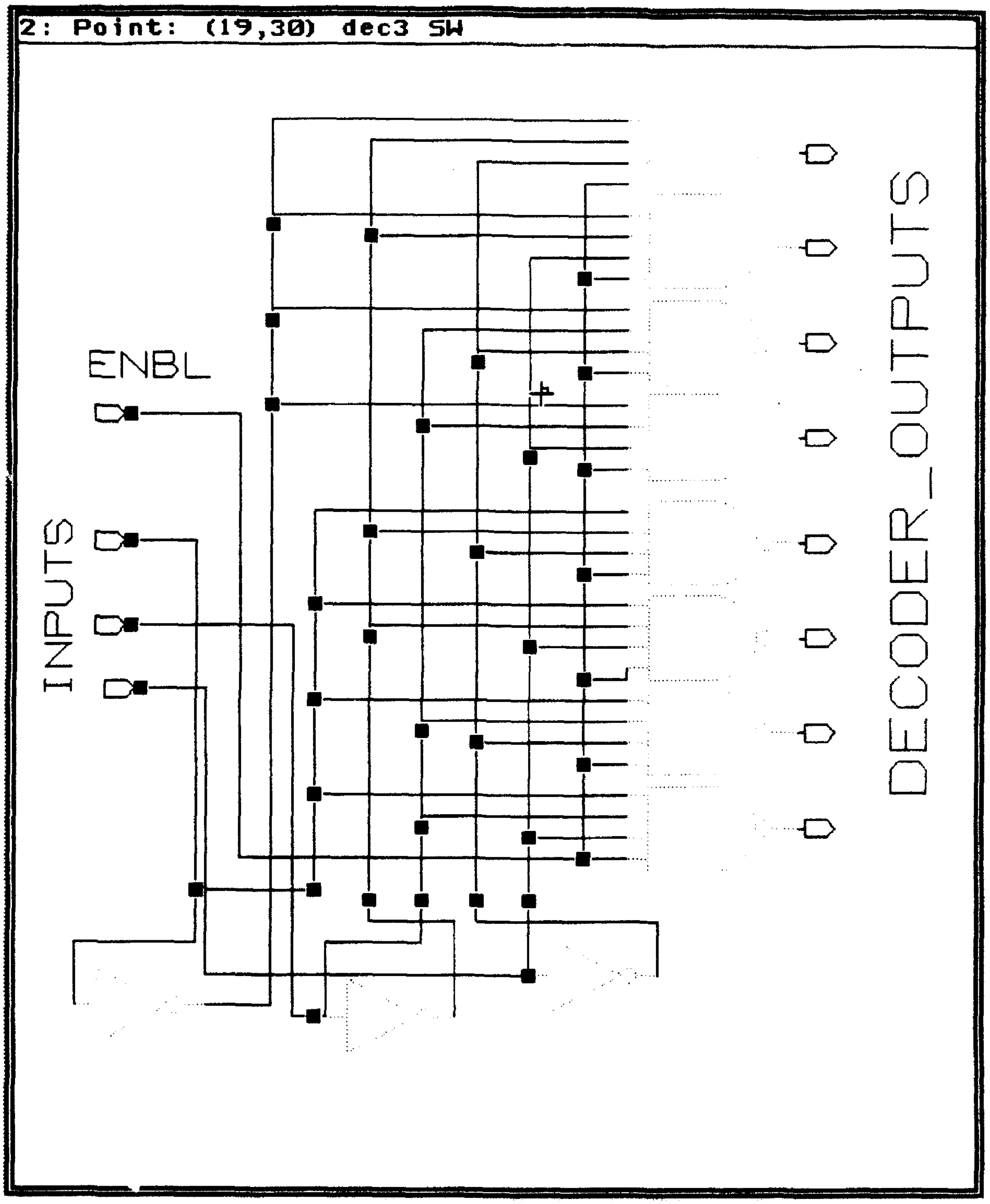


Figure 6.5 : Led Schematic of 3 to 8 decoder used in the load control circuitry

bus. As shown in Figure 6.2 the output of the latches is fed into the arithmetic unit.

6.3.2 The arithmetic unit.

The arithmetic unit has the basic function of performing a complex multiplication and accumulation. It consists of four multiplier units, an adder, a subtractor and two accumulators. The multiplier designed for the chip has a systolic array architecture[24] as shown in Figure 6.6. The multiplier is a signed binary multiplier with a 7x7 array of full adders to compute the partial products. It was decided to use an array architecture for the multiplier because for an 8 bit configuration the array architecture performs much better the other structures[25]. The final stage is a ripple adder which sums up the partial products. The sign bit is computed by a XOR gate which is fed by the sign bits of the two operands.

The input to the arithmetic unit is assumed to be signed binary because this representation is the most natural form of representing binary numbers especially at the output of a sensor array. But addition and subtraction of binary numbers can be carried out much more easily if negative numbers are in the two's complement forms. After multiplying two signed binary numbers the actual product is divided by 64 by dropping the six least significant bits. Hence the output of a 8 bit signed binary multiplier is a 9 bit result. To convert the output of the multiplier to the two's complement form an adder circuitry is added at the end of the multiplier. A demultiplexor is used to separate the positive and negative numbers. The demultiplexor is driven by the sign bit of the product. The positive numbers are sent directly to the output of the multiplying stage while the negative numbers are fed

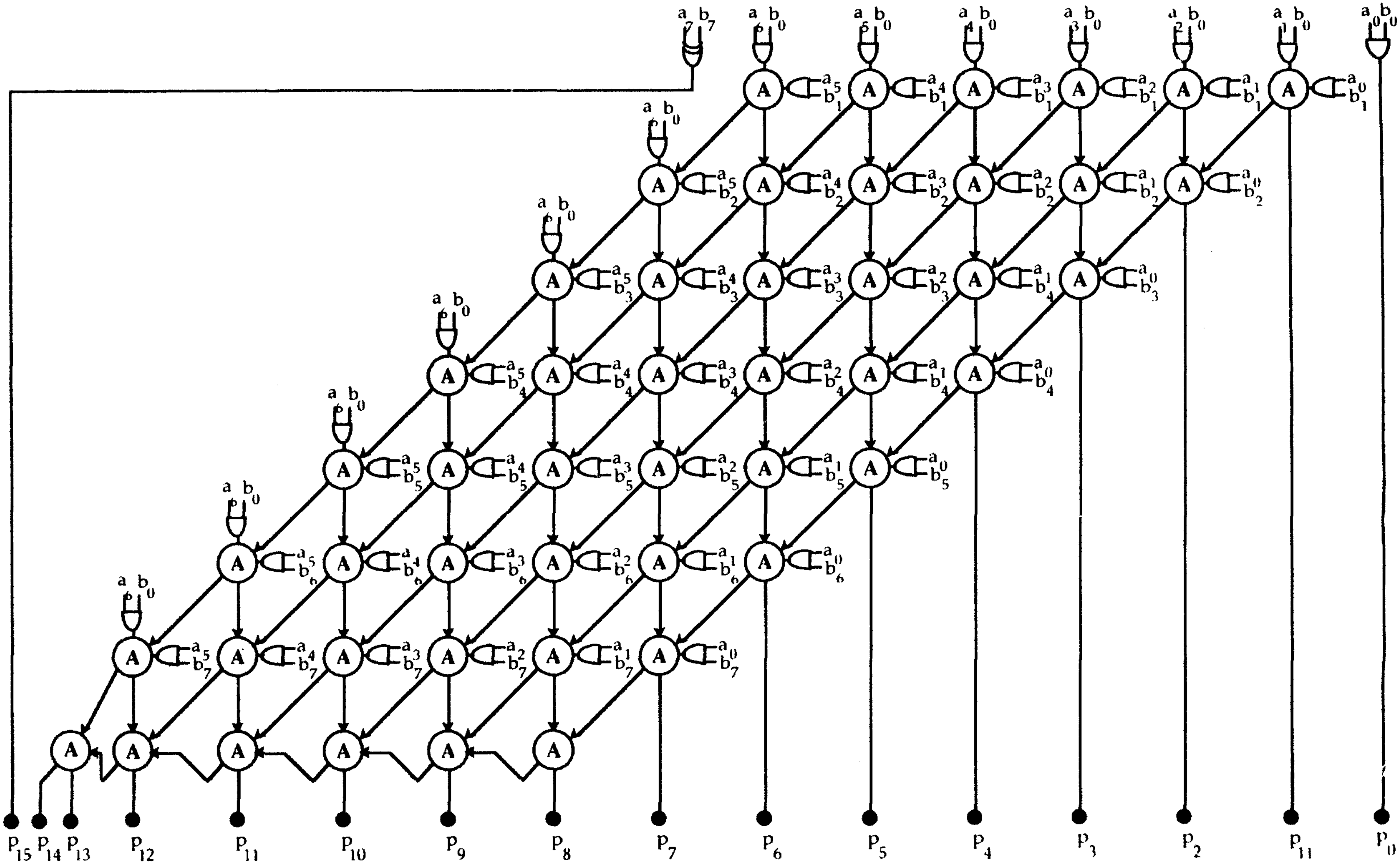


Figure 6.6 : Schematic of 8 bit signed binary systolic array multiplier used in the processor's arithmetic unit

into the two's complementing stage. The two's complementing operation is achieved by using a ripple adder and taking the complement of the input before feeding it to a full adder. The second input of the ripple adder is set to logic zero and its input carry is set to logic one. The Led schematic of the multiplying unit is shown in Figure 6.7. The Lsim adept mode simulation results of the multiplying stage are shown in Figure 6.8. The two inputs are a & b and the output is the product p.The two input numbers are +127 and -127 represented as 7f and ff in 8 bit signed binary representation. The output product is -16129 [-3f01 or -11 1111 0000 0001]. After shifting by six bits the result is -1111 1100. As this is a negative number it is represented in the 2s complement form as 1 0000 0100 or 104 in hexadecimal as shown in the figure.

After the multiplying stage, there are four such products which are the result of the first stage of a complex number multiplication operation. These are the inputs to the adder and the subtractor. Ordinarily the subtraction of two of these operands will give the real part of the result, but in the generation of a covariance matrix, a vector is multiplied by its Hermetian, which is basically the transpose of its complex conjugates. Hence the operations are reversed and an addition is performed to obtain the real part of the result. The imaginary part can similarly be obtained by subtracting the two appropriate operands.

The next stage is a 9 bit adder/subtractor. The Lsim schematic of a basic full adder circuit is shown in Figure 6.9. Figure 6.10 shows the construction of a 9 bit ripple adder generated from the basic full adder circuit. The negative operands are in the two's complement form. So addition is performed by simply adding all bits including the sign bit [26]. There is an erroneous

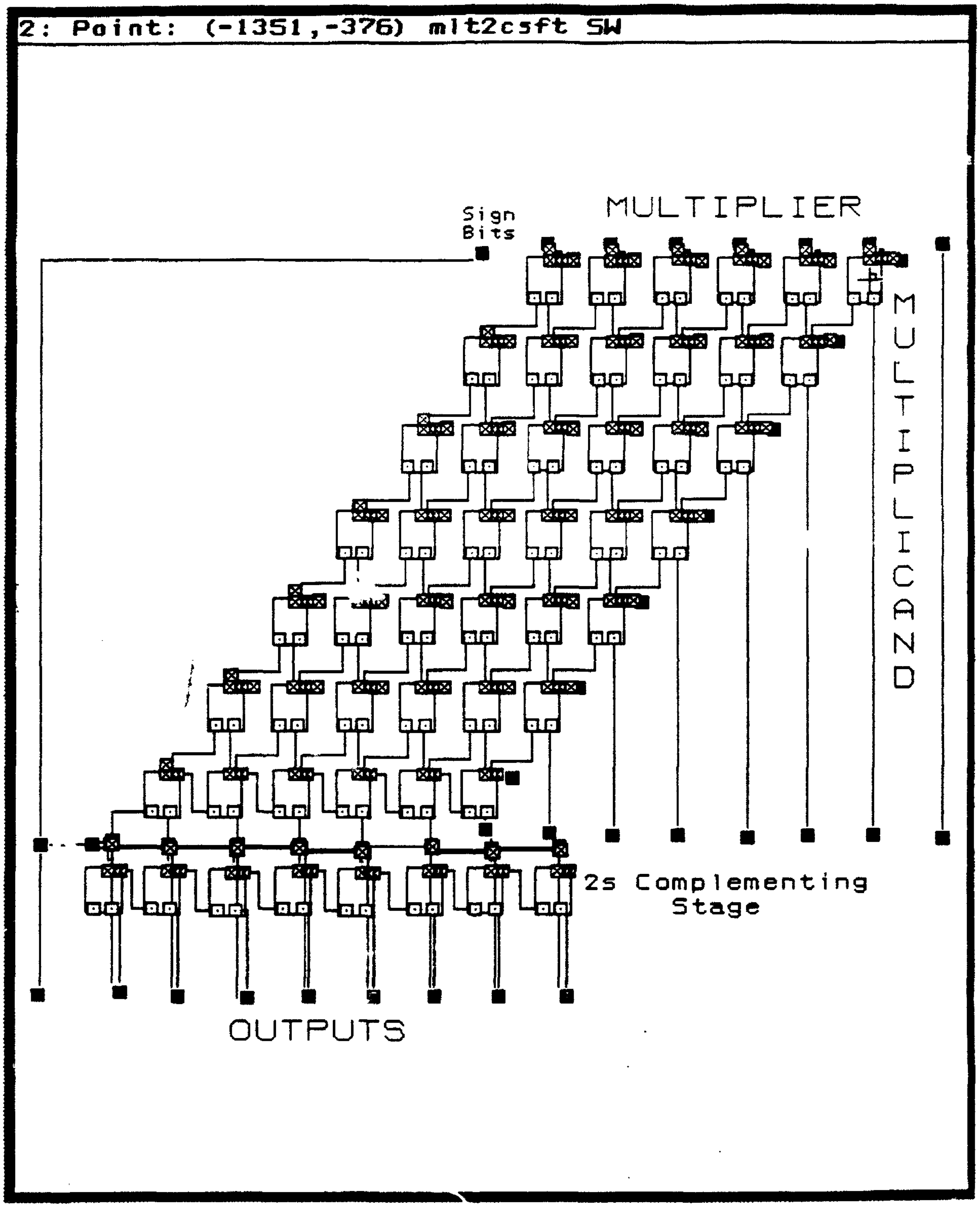


Figure 6.7 :Led top level Schematic of Multiplying stage

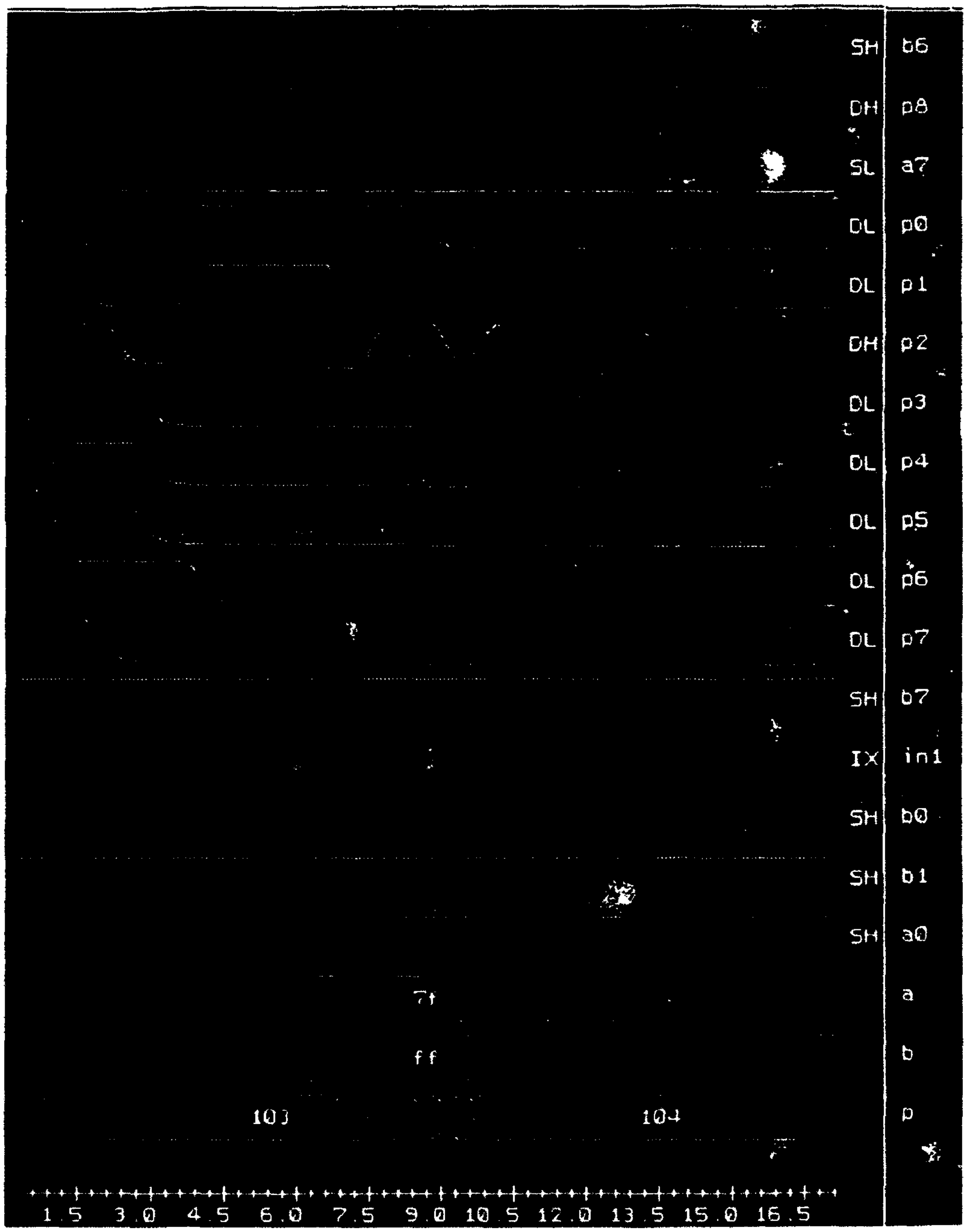


Figure 6.8 Esim adept mode simulation of the 8 bit systolic array multiplier

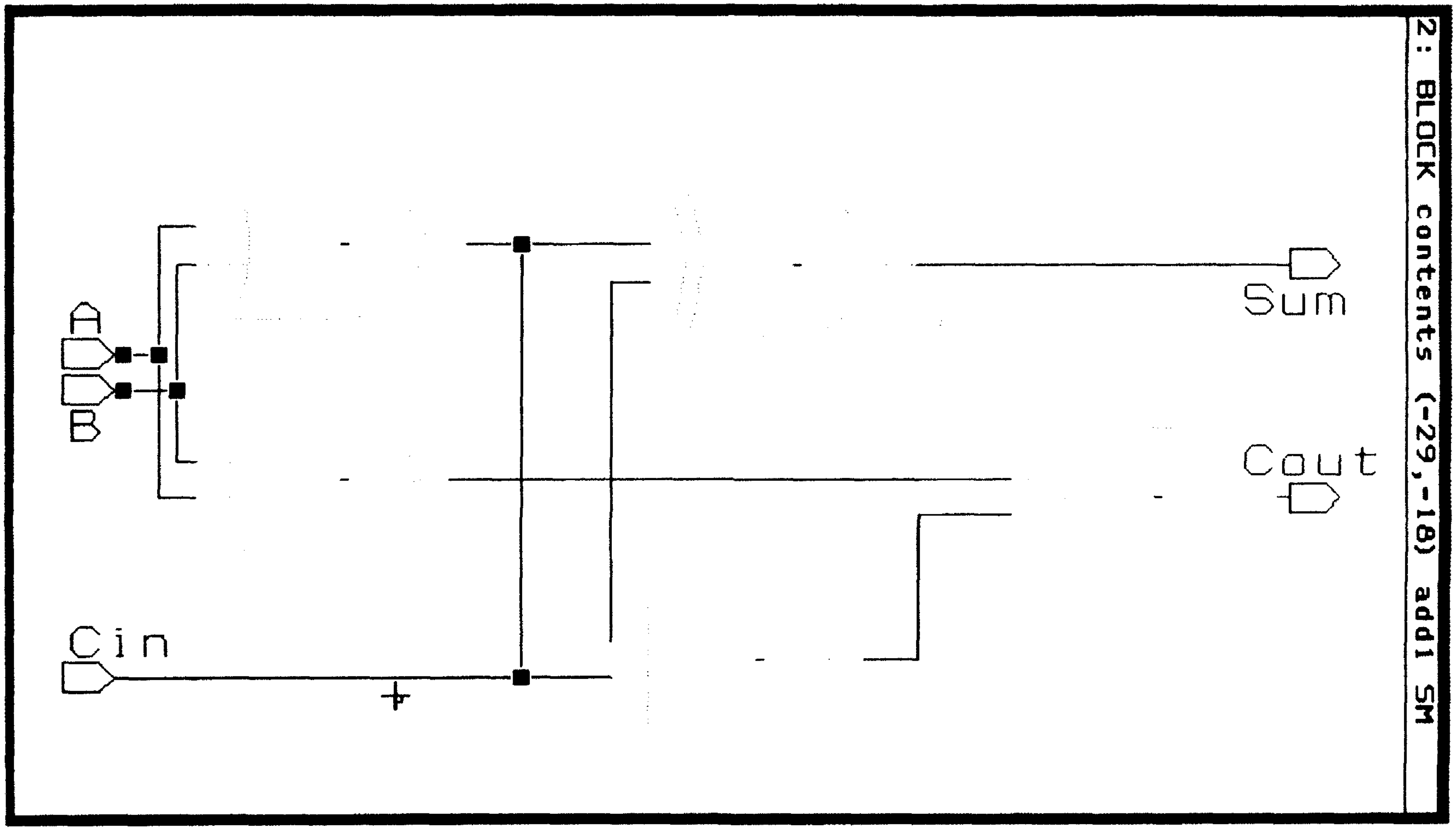


Figure 6.9 : Led Schematic of Full Adder used in the adder and accumulator stages

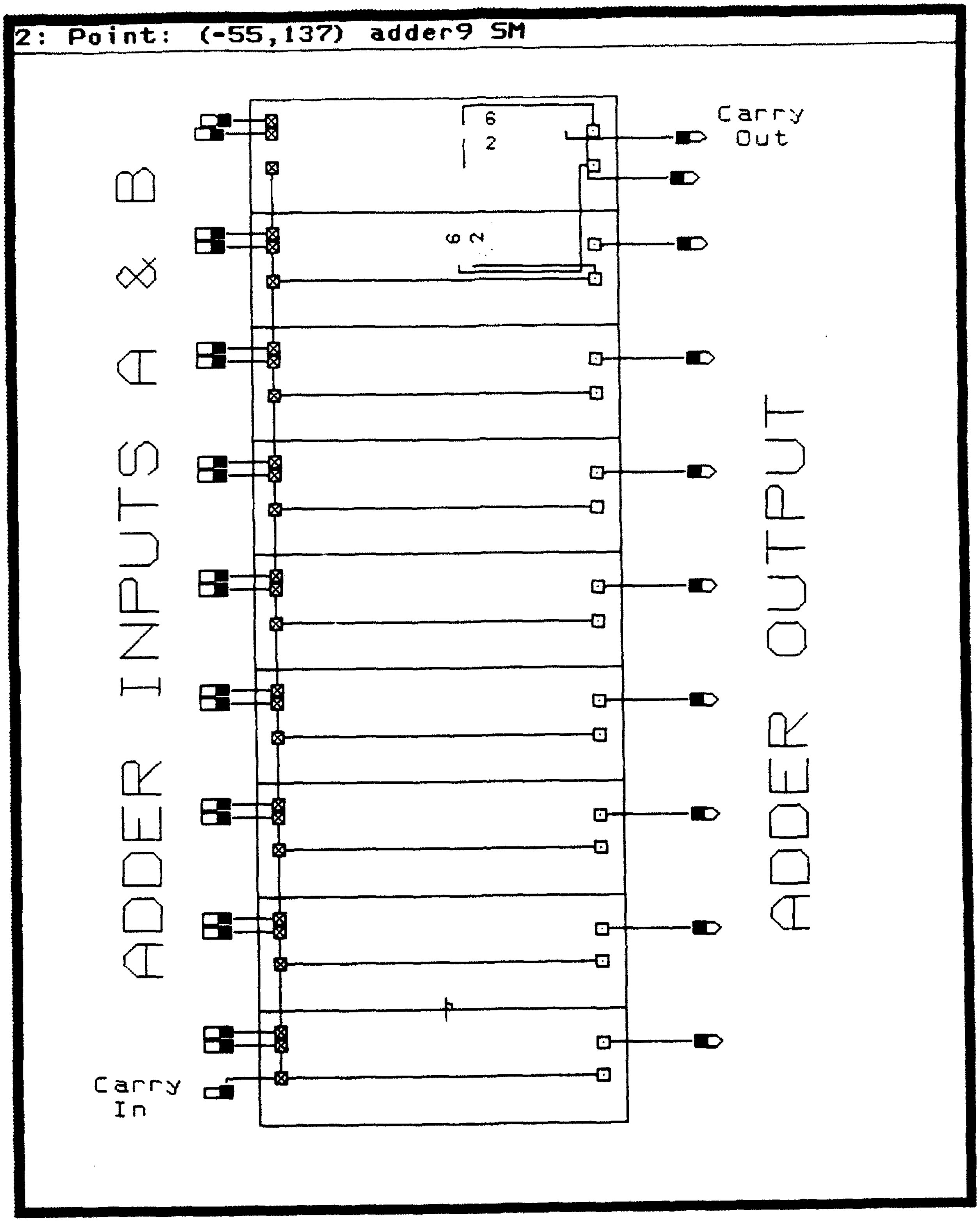


Figure 6.10 : Led top level Schematic of a 9 bit adder

reversal of the sign bit if an overflow occurs. To correct the above error, the carry into the sign bit position and the carry out of the sign bit is observed. The two carrys are applied to an XOR gate and the overflow is detected if the two carrys are not equal. Then the result is applied to another XOR gate along with the MSB to obtain the correct result. The Lsim simulation results of the adder are shown in Figure 6.11. The two operands are a and b and s is the output sum. For example the two input numbers are +80 (0 0101 0000 or 050) and +70 (0 0100 0110 or 046) and the output is +150 (00 1001 0110 or 096).

Subtraction is carried out in a similar fashion. Subtraction in two's complement arithmetic is very simple and can be achieved by taking the two's complement of the subtrahend (including the sign bit) and adding it to the minuend(including the sign bit). The basic full subtractor unit is shown in Figure 6.12 and the top level Led schematic of a 9 bit subtractor is shown in Figure 6.13. The input carry of the LSB is set to logic one and the subtrahend is complemented to achieve the subtracting operation. The overflow is once again resolved as explained previously. The Lsim results are also shown in Figure 6.14. For example when the minuend is -80 (1 1001 0000 or 1b0 in two's complement form) and the subtrahend is +70 (0 0100 0110 or 046) the difference is -150 (11 0110 1010 or 36a in two's complement form).

The last stage in the arithmetic unit is the accumulator. The accumulator basically consists of a 16 bit adder and a demultiplexing unit as shown in Figure 6.15. One operand of the accumulator is the output from the previous adder/subtractor stage. The other is the previously stored result in the memory to which the newly computed value needs to be added. The output is connected to a demultiplexing stage which places the data either on the memory bus, writing the result back to the memory or on the processor

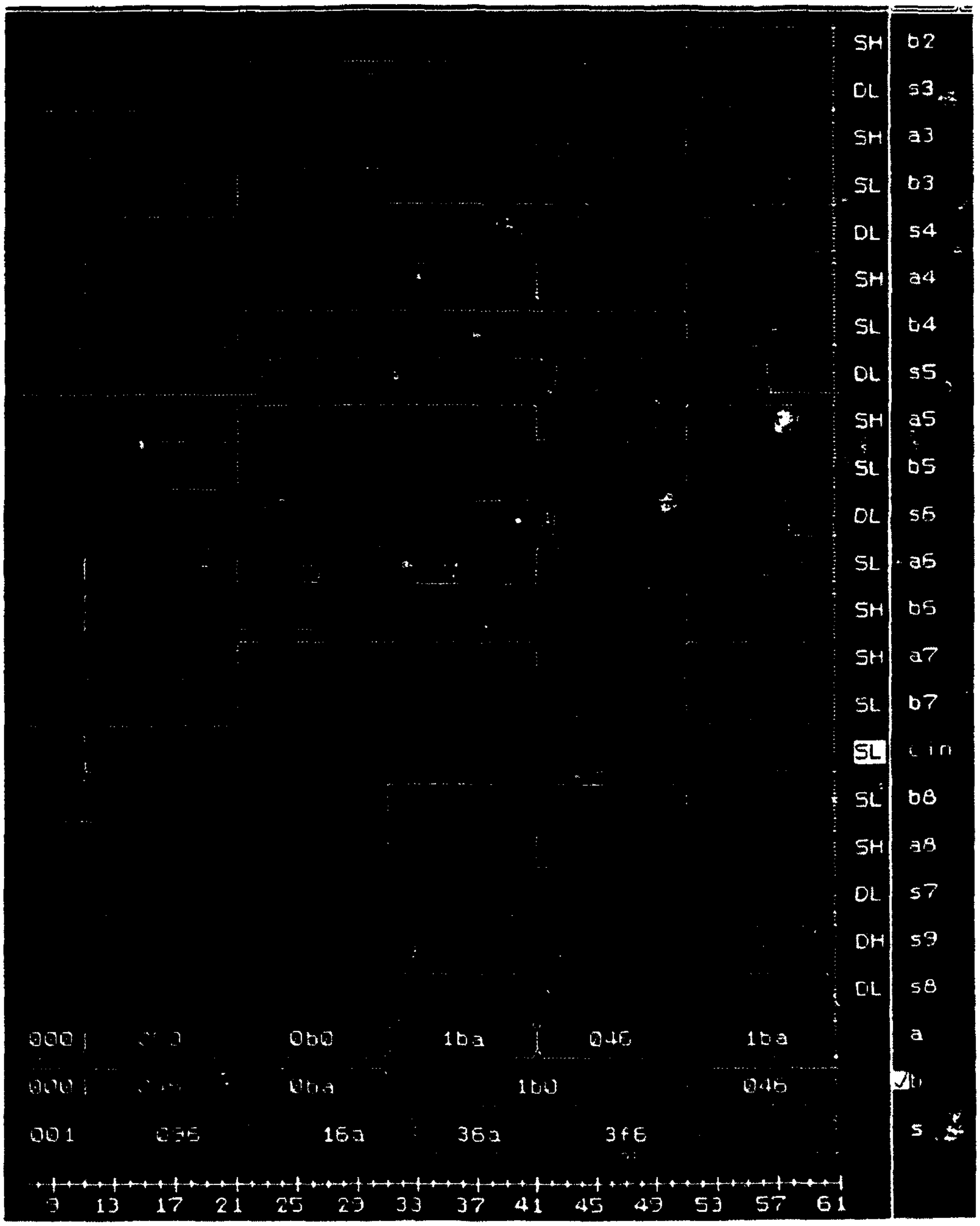


Figure 6.11 : Lsim adept simulation of 9 bit ripple adder

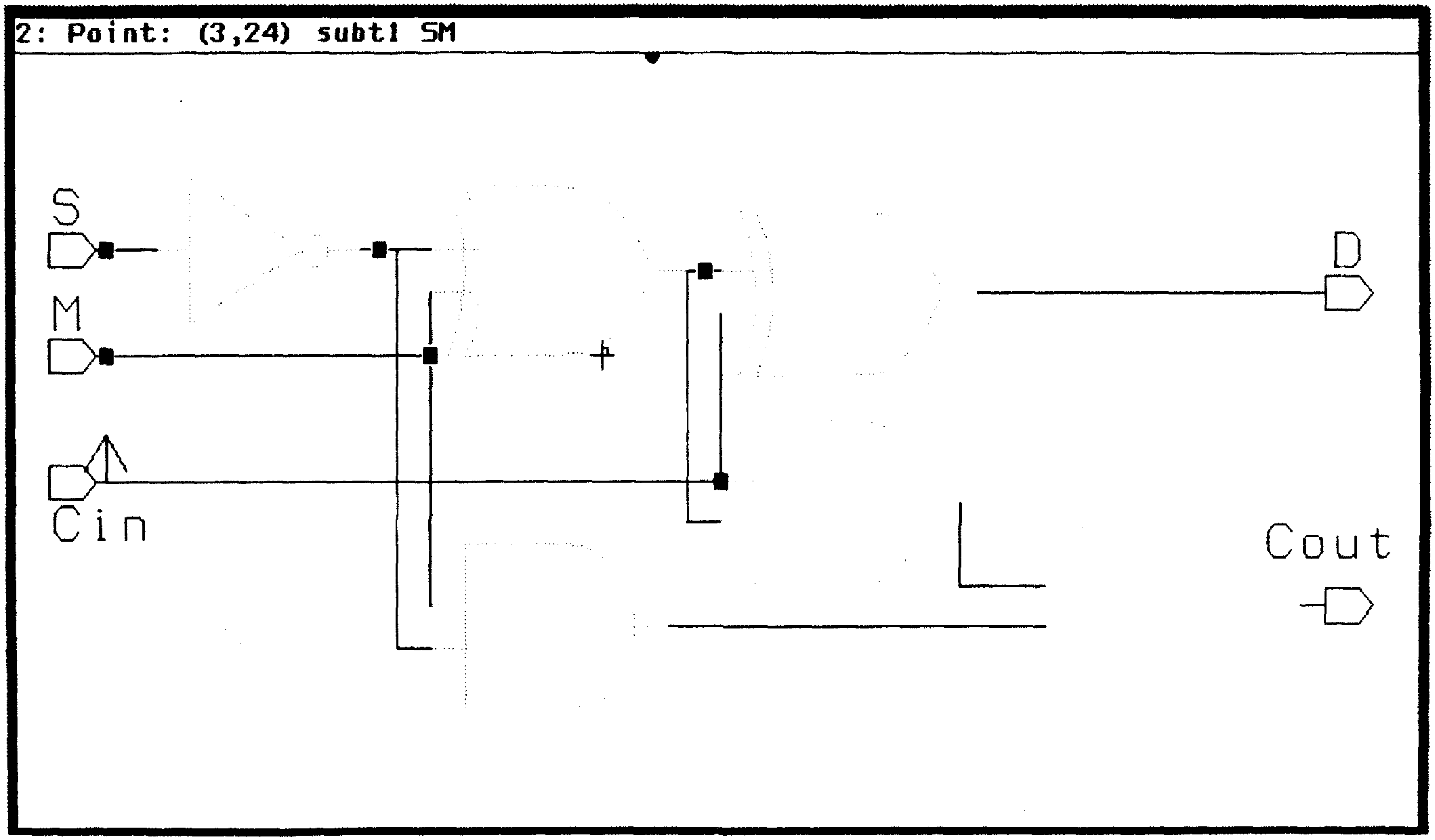


Figure 6.12 : Led Schematic of a Full Subtractor used in the 9 bit subtractor stage

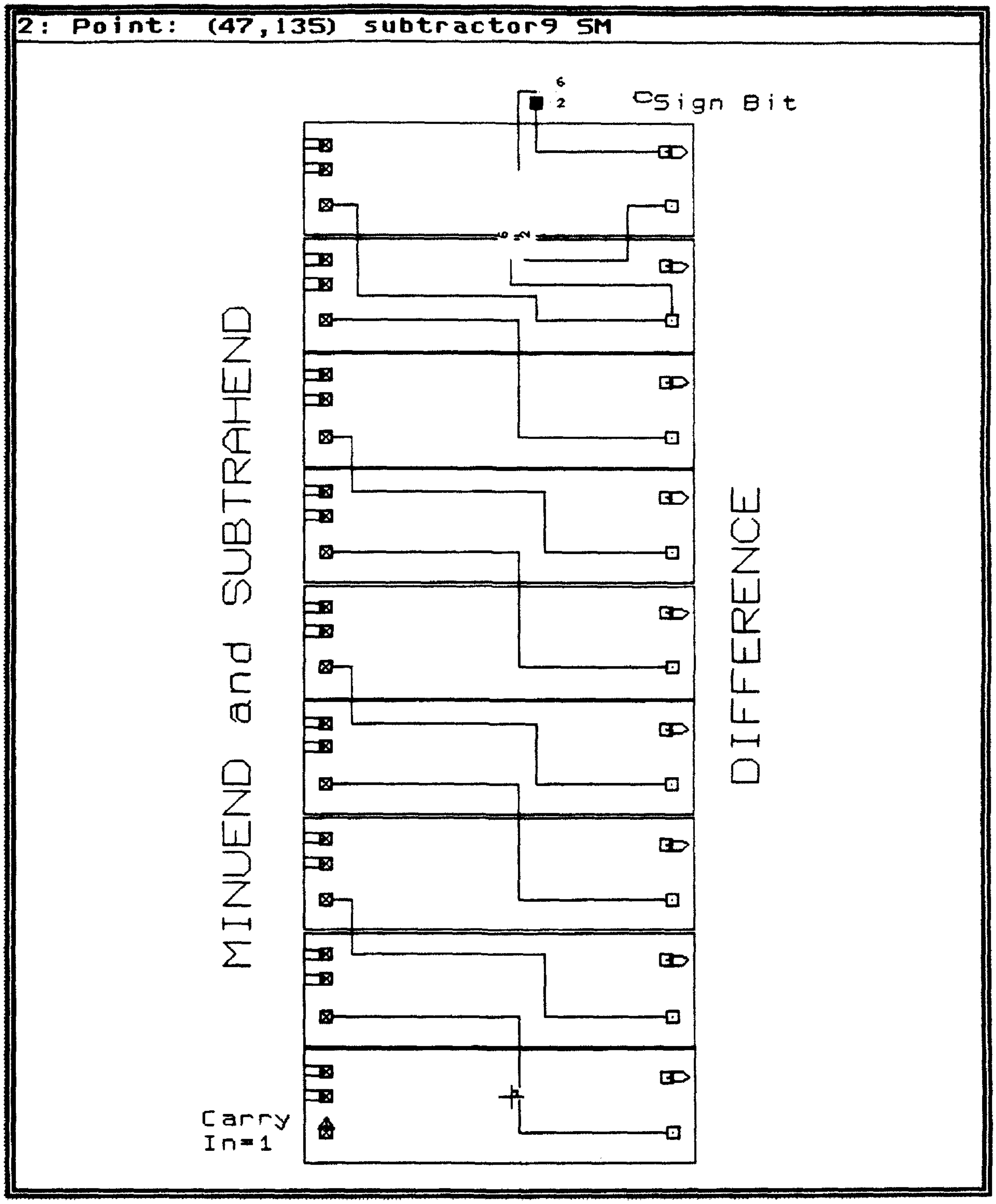


Figure 6.13 : Ledtop level Schematic of 9 bit Subtractor

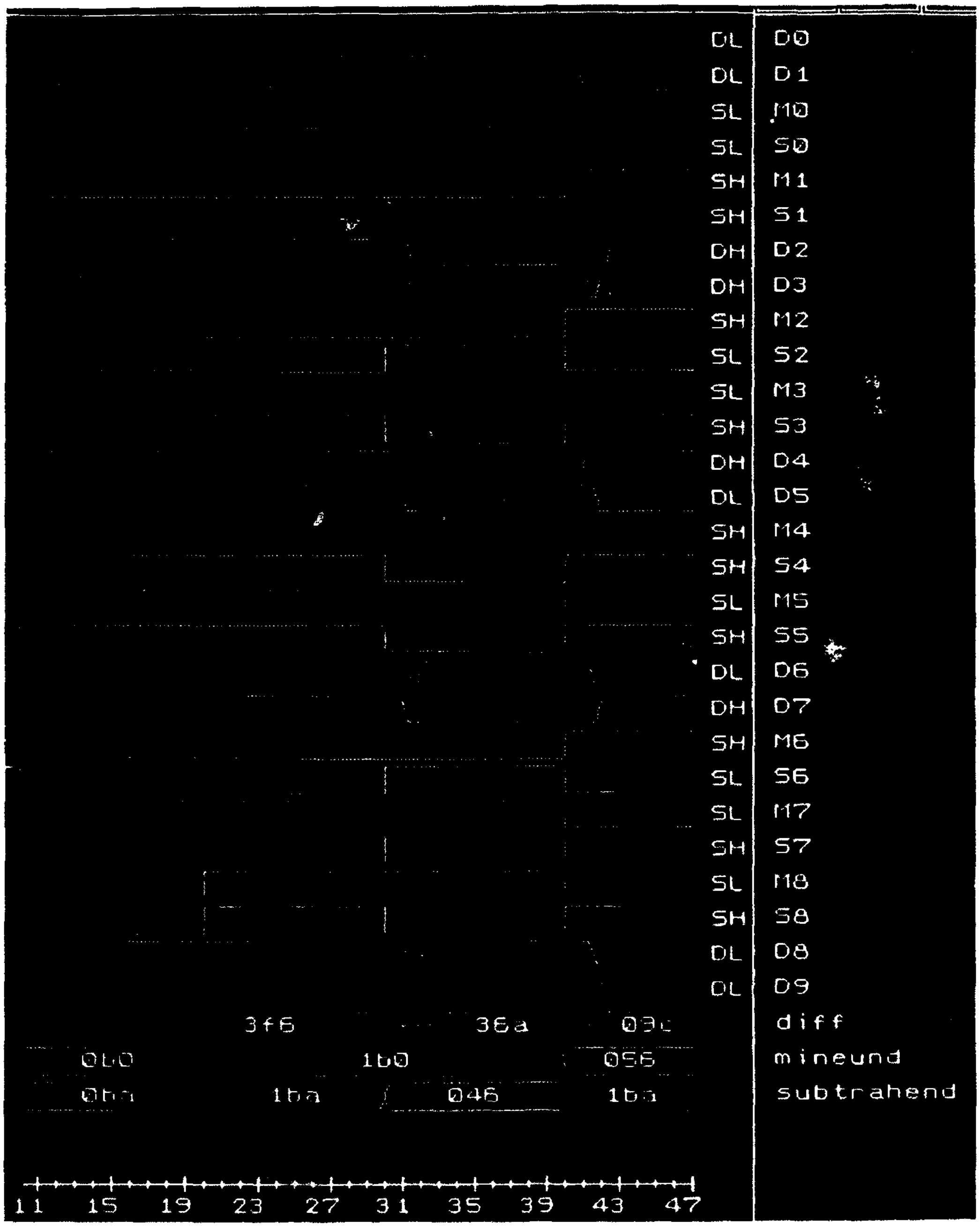


Figure 6.14 : Lsim adept simulation of 9 bit subtractor

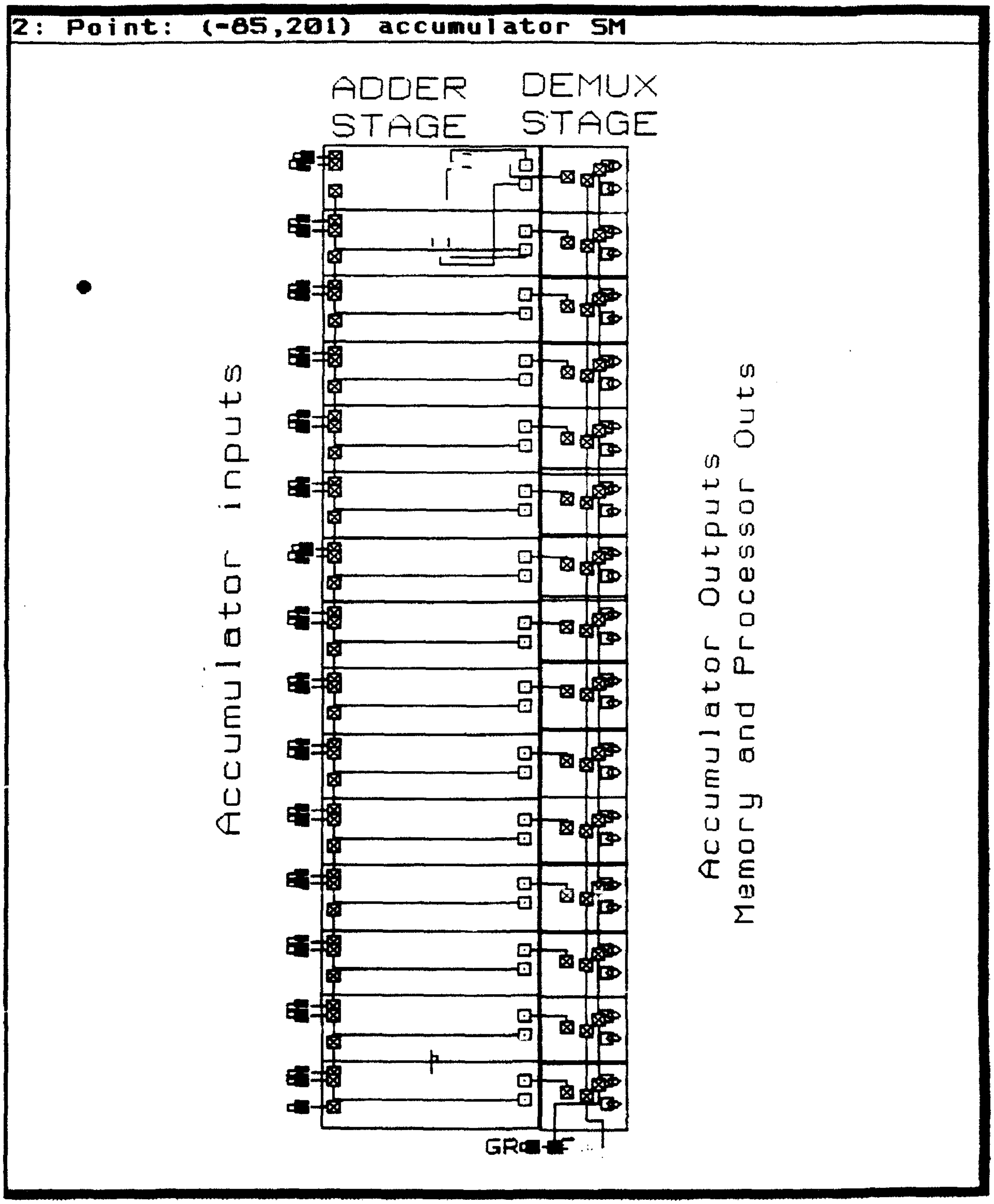


Figure 6.15 : Led top level Schematic of Accumulator stage

out bus which signifies the completion of the processing of one frame of data. The demultiplexor is controlled by the global reset signal which is obtained from the control unit. The simulation results of the accumulator are shown in Figure 6.16. As seen from the figure the inputs are a and b while the two outputs are mem (output to memory) and PR (processor output). When the global reset (GR) is pulled up the sum is put on the processor out. After it is pulled down the output is put on the mem. When the inputs are +70 (000 0000 0100 0110 or 0046) and +86 (000 0000 0101 0110 or 0056) and the input carry is set high then the result is +157 (000 0000 1001 1101 or 009d).

A major block which has been included in the schematic is the random access memory which is used to store the intermediate results of the operations. The required memory has been placed outside the chip so that a commercially available component can be used in conjunction with the processor ASIC to generate a reliable system. The memory is interfaced to the processor by a multiplexor as shown in the schematic of the processor. The data in and data out buses are connected to the multiplexor which is connected to the memory bus. The multiplexor is controlled by the input clock. The input clock has a duty cycle of 50% and hence can be used as a read/write signal. When the clock is high the processor reads from the memory and when the clock goes low the processor writes the output back to the memory. The size of the memory required for the operation is primarily dictated by the operations in the BASS-ALE algorithm which stores upto 2^9 elements during the computation of one covariance matrix. These elements are 32 bits wide including the real and imaginary components and hence require a RAM 16K bits in size. The RAM has a READ/WRITE signal, an enable and a reset signal which initializes all arrays to zero.

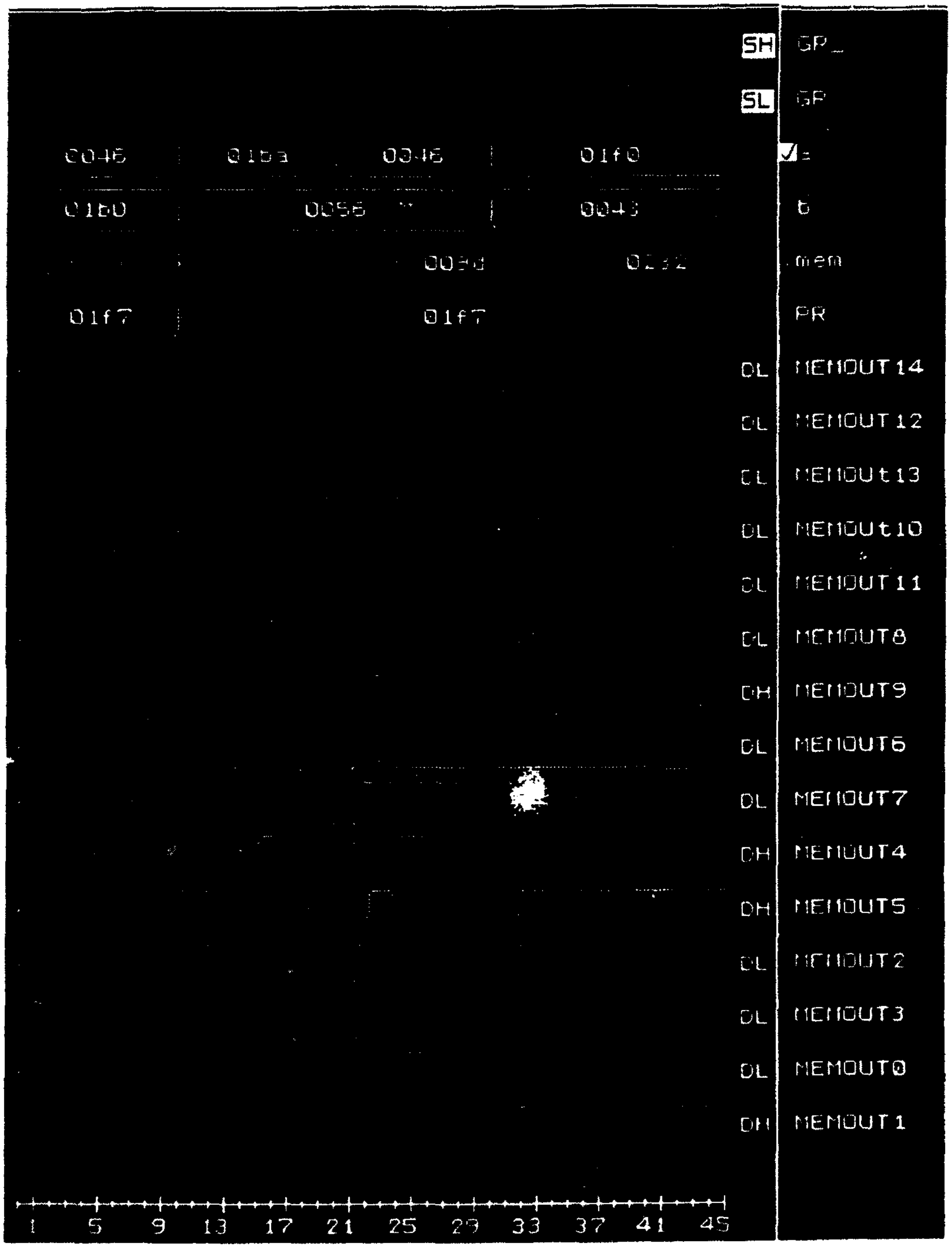


Figure 6.16 : Lumadent simulation of 16 bit accumulator

For simulation purposes M model code was written for the RAM and the Lsim simulations were carried out in the multi-level mode. The code for the M model of the RAM is shown in Appendix B.

6.3.3 The control units

The function of the control units is to generate the correct address for the retrieval of data from the memory during the accumulation stage. The control unit should also generate the global reset pulse once the processor finishes its cycle of operations. As most of the required control operation is basically to count the number of loops that the system has executed, the control unit consists mainly of counters. The counters are the asynchronous ripple type with a Master Slave T flip flop as the basic unit. A Led schematic of the MSFF is shown in Figure 6.17. The output of one flip flop is connected to the clock input of the next flip flop to generate the ripple action. The schematic of a 6 bit counter composed of the MSFF is shown in Figure 6.18.

The control unit for the narrowband MUSIC algorithm as shown in Figure 6.19 consists of two counters one of which is a 3 bit counter which upcounts to 7. This three bit counter is used to generate the address bits for the storage of the 8 different elements that are computed. The 6 MSB of the 9 bit latch are grounded, so the 8 elements will occupy the memory cells from 000000000 to 000000111. Even though there are only three address bits a 9 bit latch is used because the address bus outside the control unit is 9 bits wide. The outputs of the 3 bit counter are fed to a 3 input AND gate which generates the clock pulse for the 12 bit frame counter. The frame counter counts the 4096 loops that need to be executed during the accumulation process.

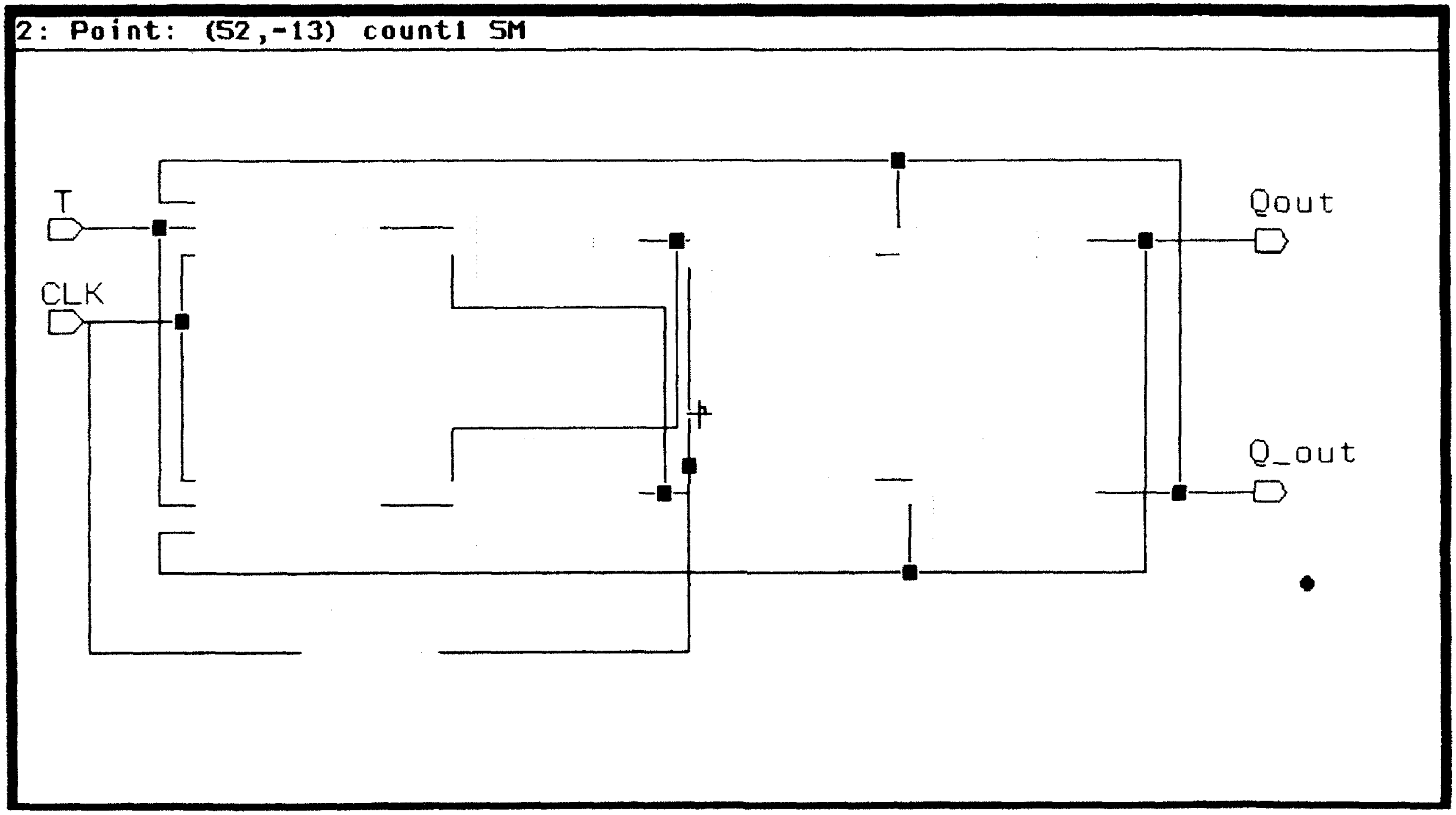


Figure 6. 17 : Led Schematic of a T type master slave flip flop used in the ripple counters

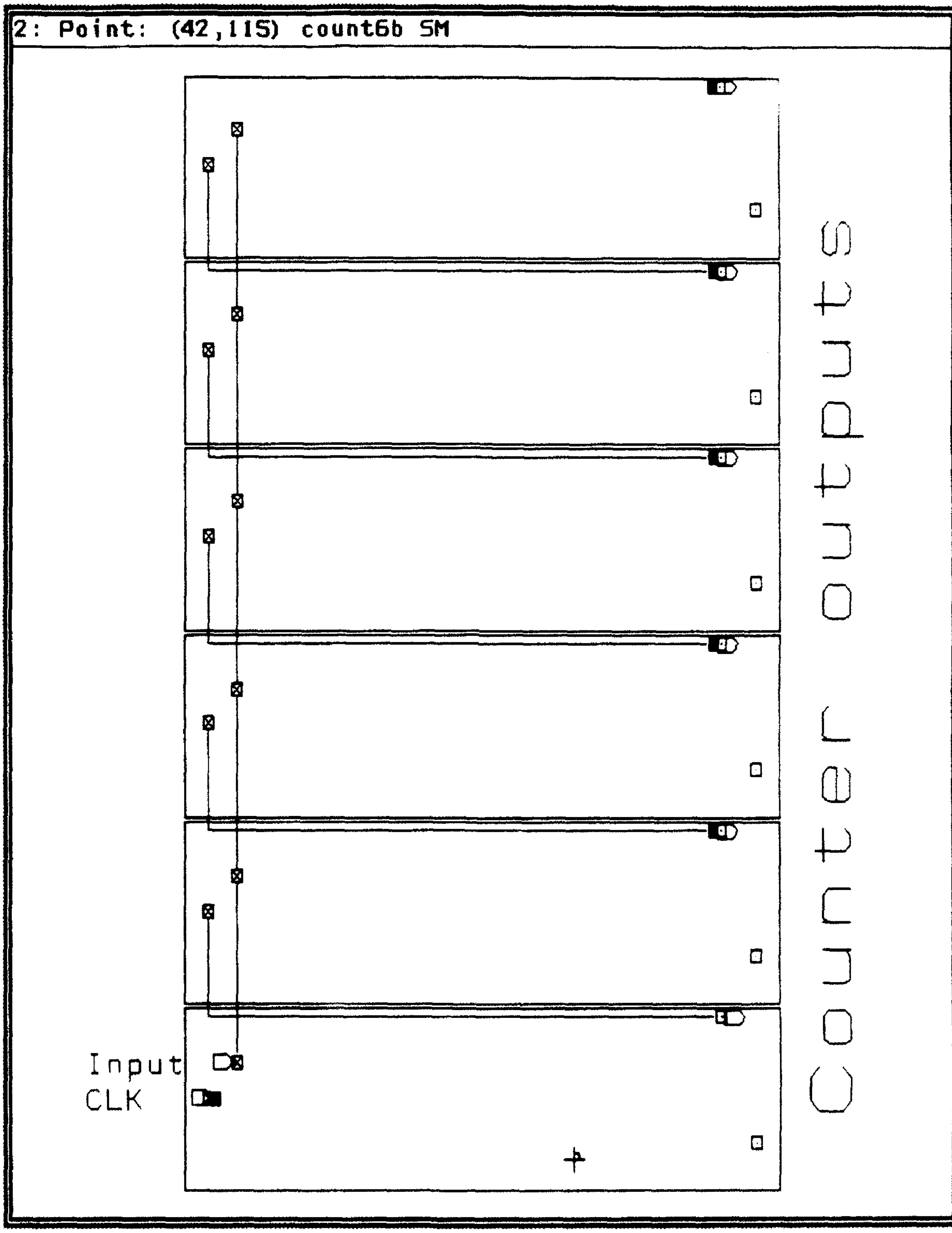


Figure 6.18 : Led Schematic of 6 bit counter used in the control circuitry

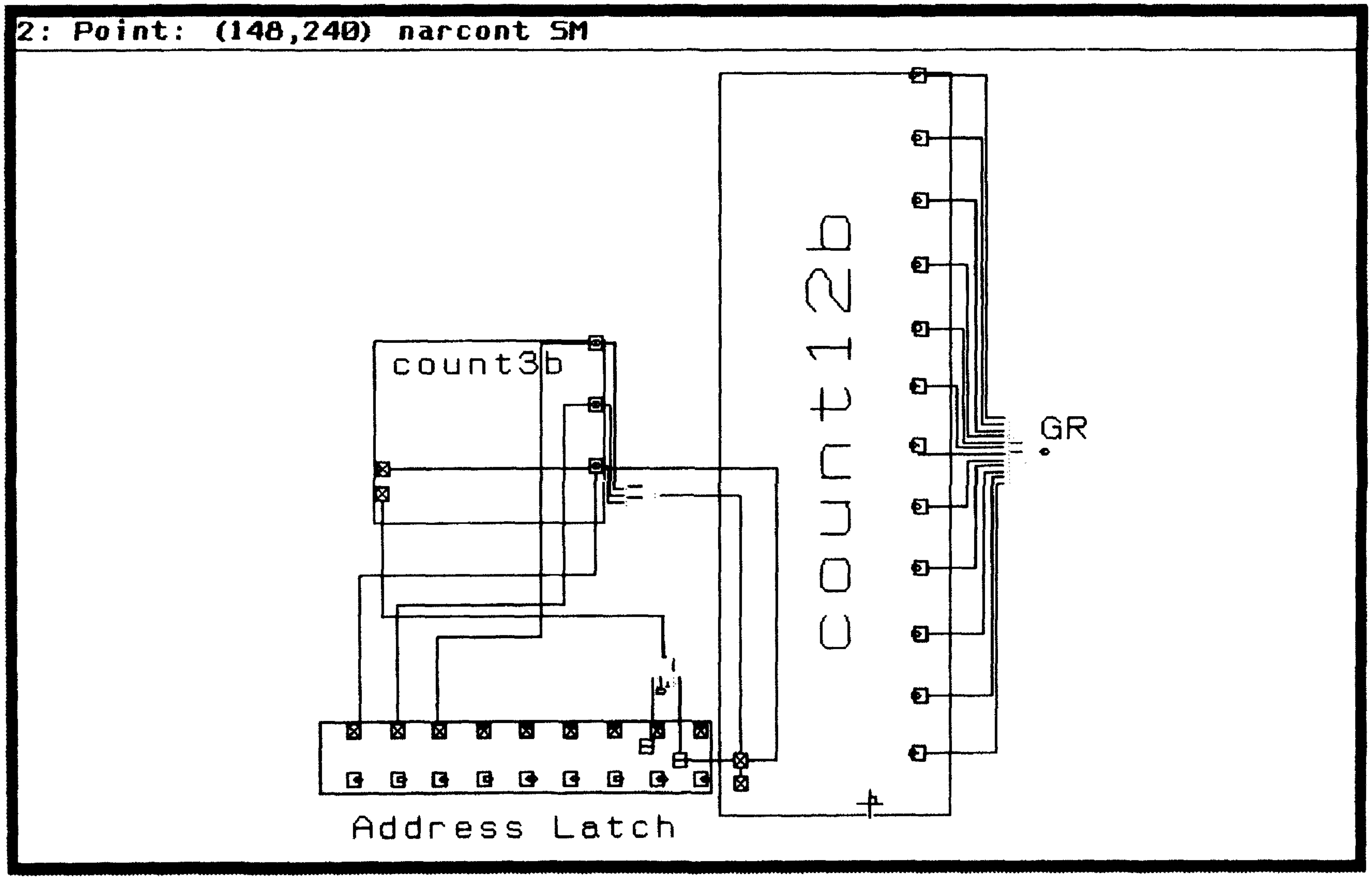


Figure 6.19 : Led Schematic of control unit for Narrowband MUSIC Algorithm

The control unit for the BASS-ALE algorithm as shown in Figure 6.20 has four counters, three of which are used to generate the address bits. The first counts the number of elements in the column, the second counts the column number in the micromatrix while the third keeps track of the micromatrix number in the submatrix. The required memory is 2^9 and the address runs all the way from 000000000 to 111111111. The 9 bit counter controls the numbers of frames which the processor needs to accumulate which in this case is 512.

The control unit for the bilinear transformation algorithm as shown in Figure 6.21 has a 3 bit element counter to count the element number in the column. The next one a 6 bit counter, is used to count the 33 frequencies and hence upcounts from 0 to 32. Once it reaches 32, the logic circuitry (which is a *NOR gate with an inverted MSB) resets it to zero and clocks the 6 bit frame* counter. The frame counter counts the 64 frames that need to be accumulated.

Once all the modules were individually simulated they were called as instances into the top level processor cell in Led and connected. The netlist was generated and an Lsim simulation was run on the netlist. The results of the simulations are shown in the Figure 6.22. The inputs to the multipliers are mltina, mltinb, mltinc and mltind. Two of these are the imaginary and real parts of the **X** input while the other two are the elements of the **Y** vector. The outputs of the complex multiplication are add and sub, while acc1 and acc2 give the values after accumulation. The input to the two accumulators from the memory are given by mem1 and mem2. The processor was simulated over two cycles and the multiplication and accumulation operations were verified.

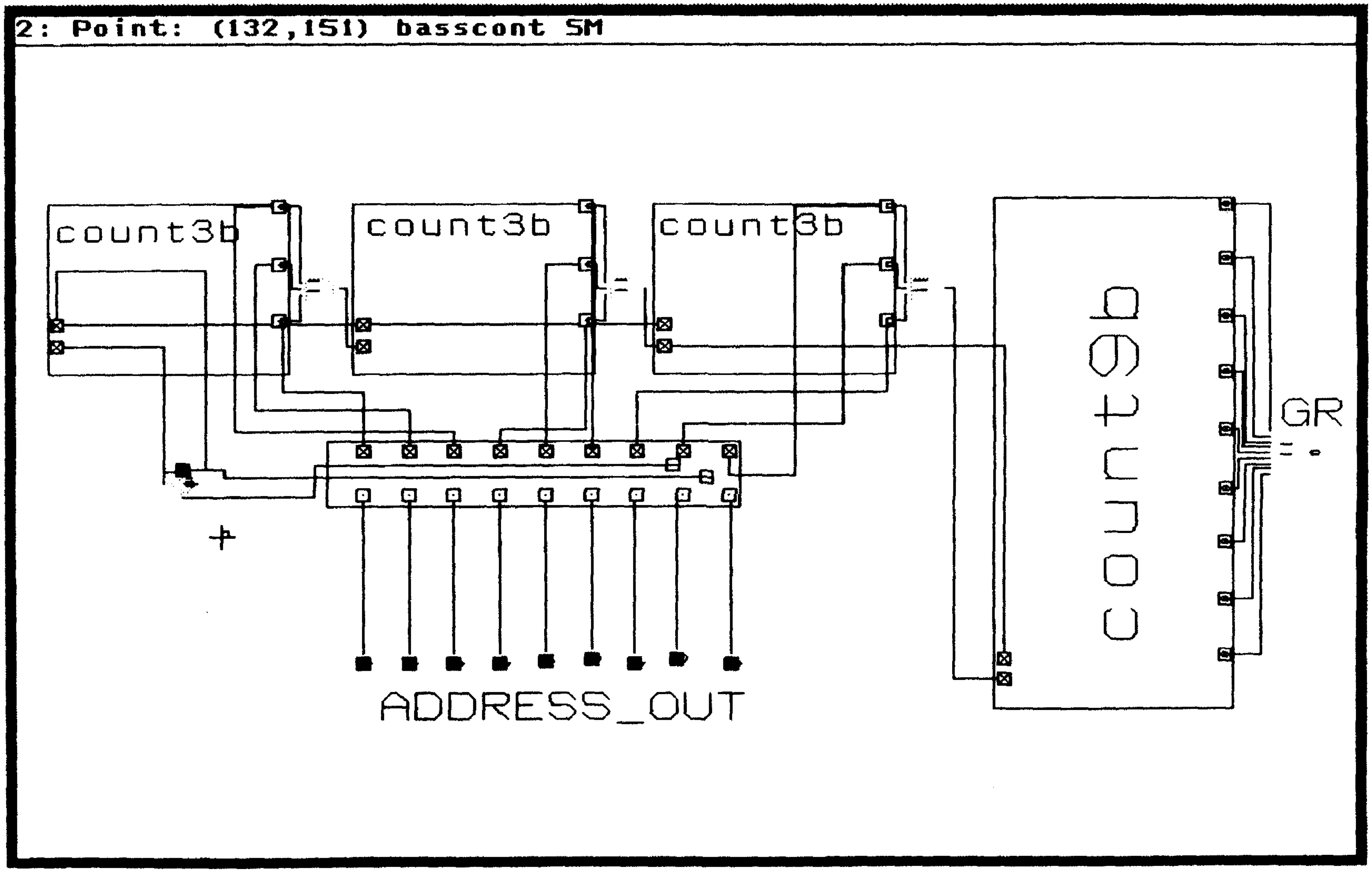


Figure 6.20 : Led Schematic of control unit for the BASS-ALE algorithm

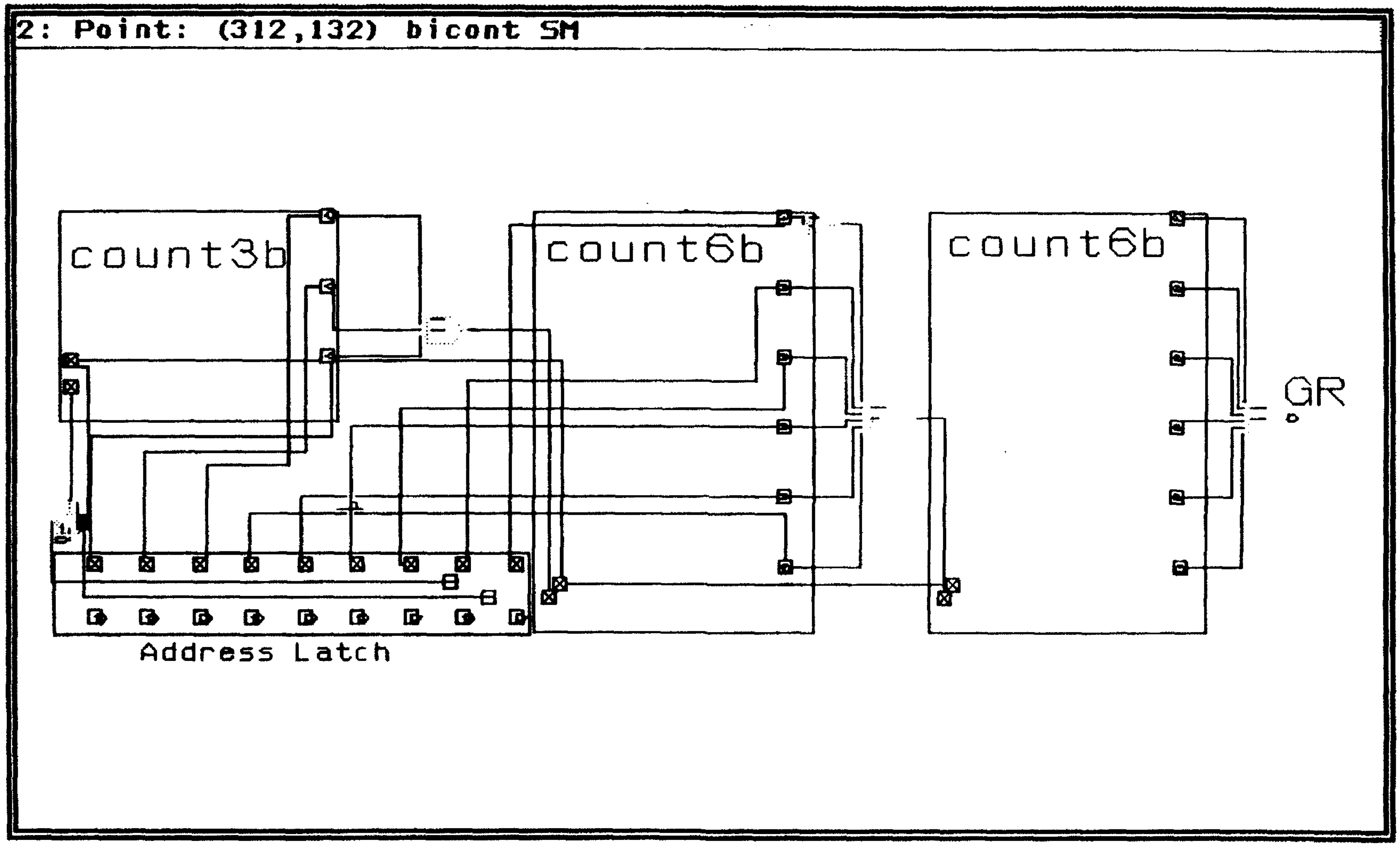


Figure 6.21 : Led Schematic of control unit for Bilinear Transformation algorithm

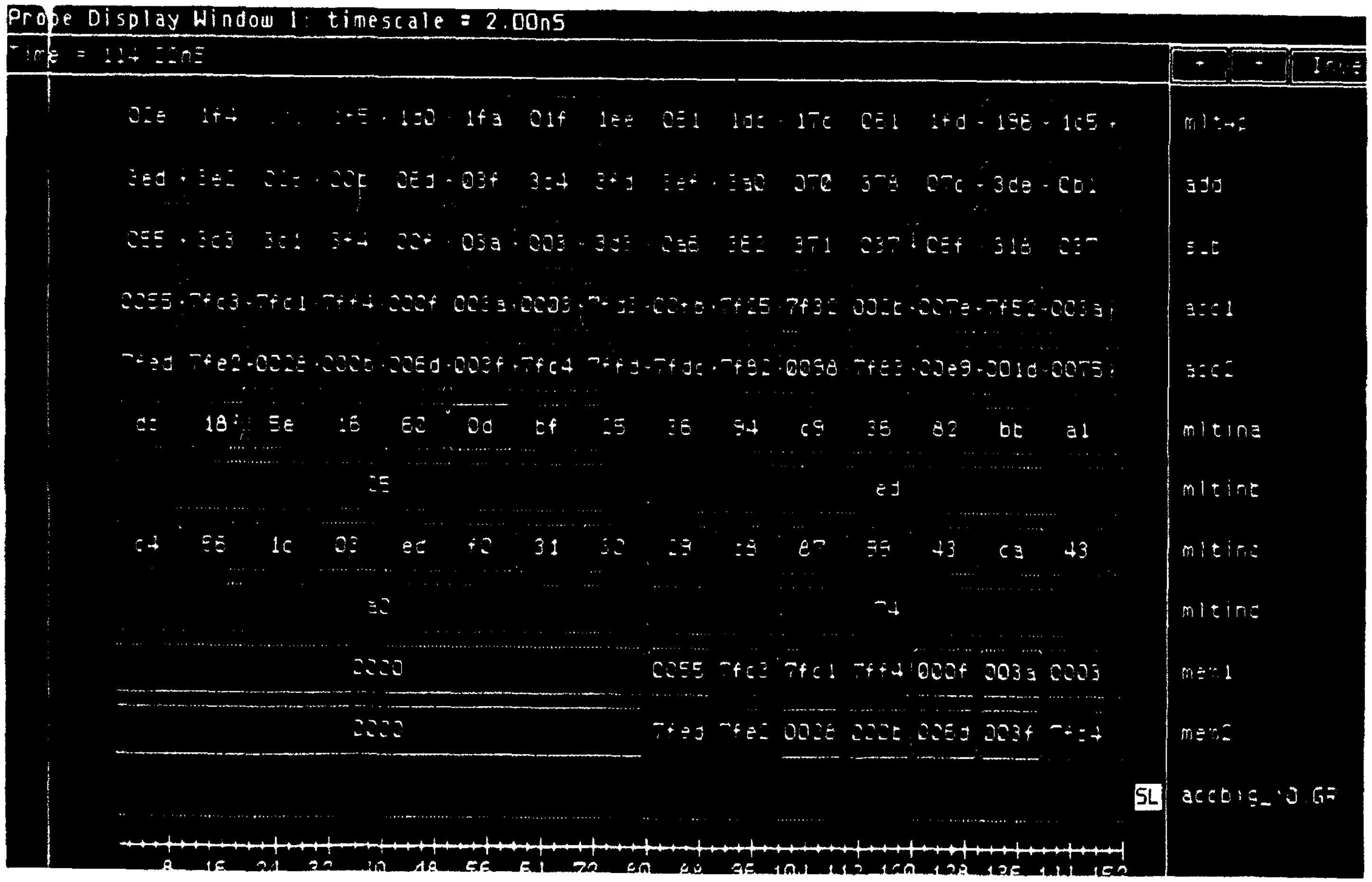


Figure 6.22 Lsim simulation results of the covariance matrix processor

A particular computation is given as an example below:

Consider the case in the second cycle where the two elements multiplied are 54 + i41 and -109 + i116. They can be represented in hexadecimal as 36H + i29H and -6dH + i74H. They are given to the four multiplier inputs as mltina, mltinc, mltinb and mltind as the eight bit signed binary numbers 36,29,ed and 74 respectively as shown in Figure 6.22. The outputs of the multiplier are shown below:

Product	Before shift	After 6 bit shift
axb = 36 x -6d	1 01 0110 1111 1110 (-16fe)	1 0101 1011 (-5b)
cxd = 29 x 74	0 01 0010 1001 0100 (1294)	0 0100 1010 (4a)
bxc = 29 x -6d	1 01 0001 0111 0101 (-1175)	1 0100 0101 (-45)
axd = 36 x 74	0 01 1000 0111 1000 (1878)	0 0110 0001 (61)

The next stage is the adder/subtractor stage. The calculations are

ab + cd = -5b + 4a = -11(1 0 0001 0001) or (1 1 1110 1111 or 3ef in 2's complement)

ad - bc = 61 - (-45) = a6 (0 0 1010 0110 or 0a6)

The adder output (3ef) which is the real part of the product and the subtractor output (0a6) which is the imaginary part are shown in the figure (add & sub signals). These signals are one of the inputs to the accumulator stage.

The two signals mem1 and mem2 are the other input to the accumulator stage. These are the accumulator outputs (acc1 and acc2) from the previous cycle which are read in from the memory The accumulator calculation is shown below:

add + mem2 = acc2 (negative numbers are in 2's complement form)

3ef (1 1 1110 1111 or -11) + 7fed (1 11 1111 1110 1010 or -16)

= 7fdc (1 11 1111 1101 1100 or -27)

sub + mem1 = acc1

0a6 (0 0 1010 0110) + 0055 (0 00 0000 0101 0101) = 00fb (0 00 0000 1111 1011)

The processor function can be verified from the above calculations.

The netlist for the processor was generated from the Led schematic. Then AutoCells was used to generate the layout of the processor. Figure 6.23 shows the layout of the complete processor. The layout was verified by simulating the netlist for the whole processor. The terminal were placed so that the routing to the pins can be done very easily. The data input terminals and the input control signals are placed at the top. The data bits (memory and processor out) are placed at the sides and the address bits are placed at the bottom. The processor was layed out in 25 rows and the total area was approximately 2200 x 5800 μm^2.

The pin diagram for the ASIC is shown in Figure 6.24 The chip will fit in a 120 pin frame available through MOSIS. The pin designation is according to the terminal placements in the layout The data pins are: -

In0 - In7	- Real part of input element
In8 - In15	- Imaginary part of input element
Out0 - Out15	- Real part of processor output element

Out16 - Out31	- Imaginary part of processor output element
Mem0 - Mem15	- Real part of memory element
Mem16 - Mem31	- Imaginary part of memory element

The I/O diagram of the processor is shown in Figure 6.25. The data pins are connected to the memory as shown. The address bits are supplied from the processor and a global reset pin is supplied so that the memory chip can be reset after one cycle of computations.

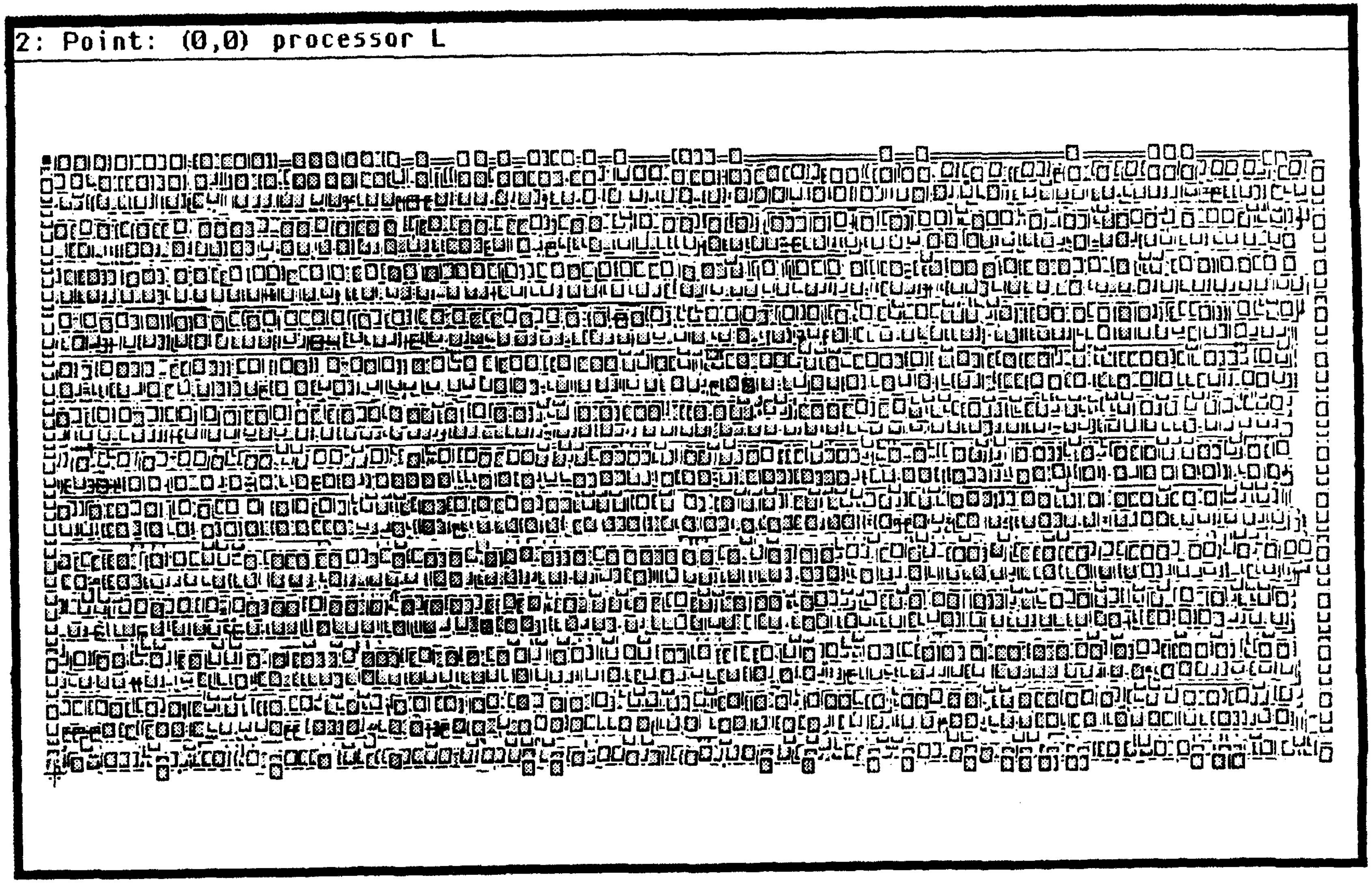


Figure 6.23 : Layout generated using AutoCells for the covariance matrix processor

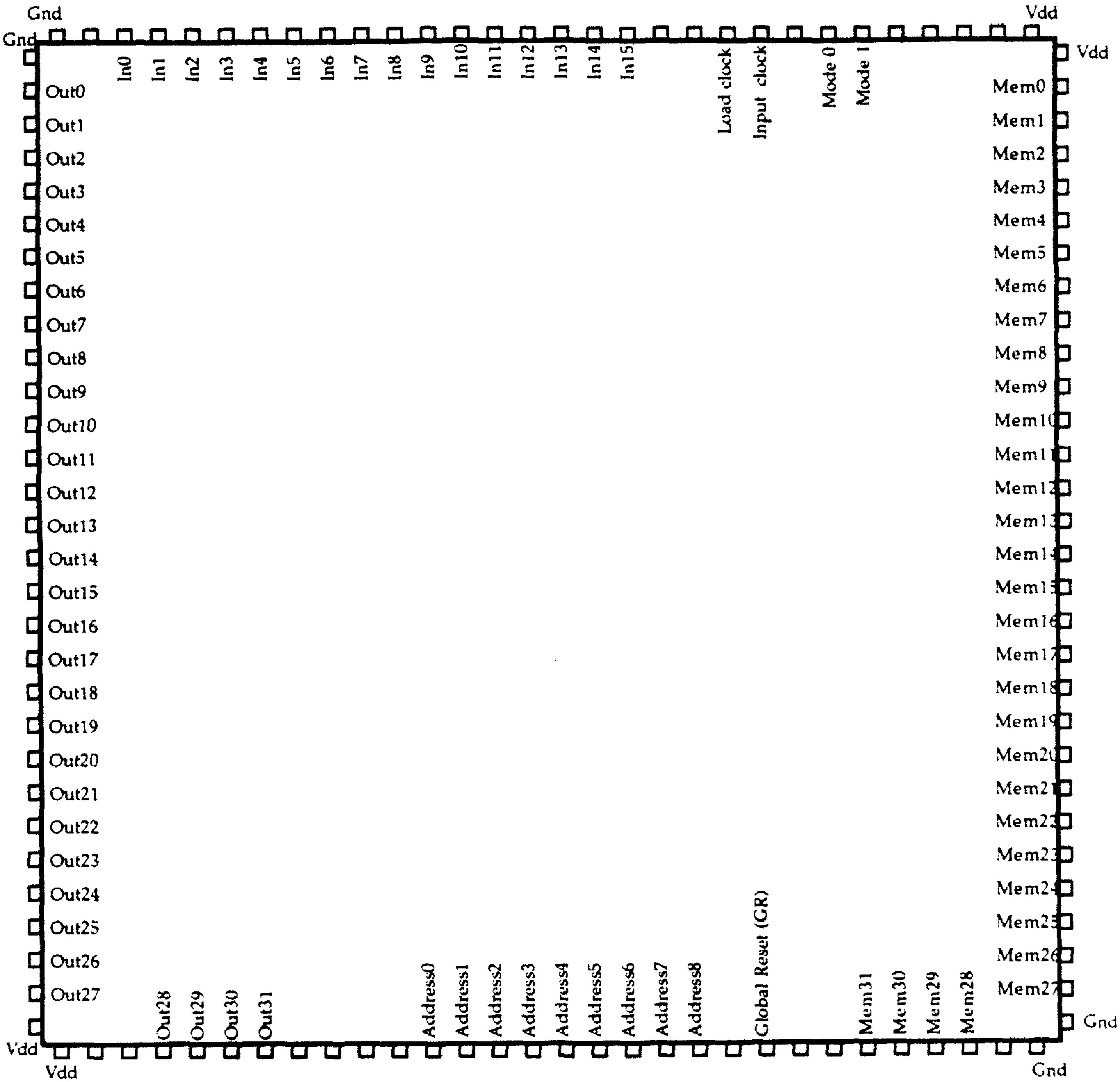


Figure 6.24 : Pin diagram for the combined covariance matrix processor on a standard 120 pin frame available through MOSIS

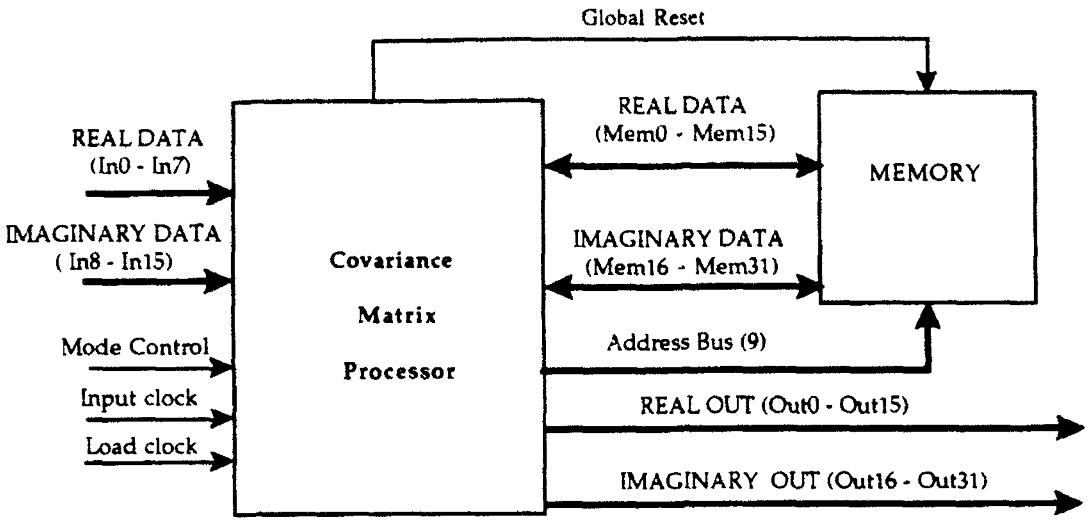


Figure 6.25 : I/O diagram of the covariance matrix processor

CHAPTER 7

Conclusions

This thesis work has dealt with the problem of estimation of direction of arrival of signals at a sensor array. In particular the wideband problem has been investigated and the bilinear transformation algorithm has been explored and an implementation devised. The algorithm was modified so that all the possible parallelization and pipelining can be fully exploited. The algorithm is completely modularized so that each module can be designed and simulated independently of each other.

A fully parallel and pipelined architecture for the algorithm is described. The entire architecture requires multiprocessor structure so the system is designed using multiple modules. The system architecture consists of commercially available components like the DSP 56000 for the FFT stage and the ASIC chips designed for other modules. Various modules were designed with emphasis on the timing requirements and the simplified routing of data which are the prime necessities for a system operating in real time.

A combined covariance matrix processor has been implemented using 0.8μm CMOS technology. Two other DOA estimation algorithms have been selected namely the narrowband MUSIC algorithm and the broadband BASS-ALE method. One common step in all these algorithms is the computation of the covariance matrix and hence a combined processor has been developed which will perform this stage of operation for all the three algorithms.

The processor has been simulated at the VHDL level using Powerview and then at the transistor level using GDT Lsim. The construction of the processor was done using the Lxcells utillity in GDT. Finally the processor was layed out using GDT AutoCells.

APPENDICES

Appendix A

VHDL programs for various modules in Powerview

```
----------------------------------------------------------------------
-- Behavioral Model for 16-Bit ACCUMULATOR.
--
-- Inputs: A leg (16 bits)
-- B leg (16 bits)
-- Carry in (1 bit)
-- Result register clock
-- Global Reset
-- Outputs: Memory data out (16 bits, latched)
-- Processor data out (16 bits, latched)
-- Memory carry out (1 bit, unlatched)
-- Processor carry out (1bit, unlatched)
----------------------------------------------------------------------

----------------------------------------------------------------------
-- Interface declaration:
----------------------------------------------------------------------

entity accumulator is

-- Generic delays, with default values

 generic (cout_delay: time := 6500 PS; -- Carry out delay
 reg_delay: time := 6000 PS); -- Register delay

-- IO ports:

 port (signal mout: out vlbit_vector(0 to 15);
signal pout: out vlbit_vector(0 to 15);
 signal mcout: out vlbit;
 signal pcout: out vlbit;
 signal a,b: in vlbit_vector(0 to 15);
 signal cin: in vlbit;
 signal GR: in vlbit;
 signal clk: in vlbit);

end accumulator;

----------------------------------------------------------------------
-- Architecture body:
----------------------------------------------------------------------

architecture behavior of accumulator is

signal accout: vlbit_vector(0 to 16); -- ACCUMULATOR output: cout & dout
signal ResReg: vlbit_vector(0 to 15); -- Result register
```

```
begin

------------------------------------------------------------------
-- Concurrent ACCUMULATE process statement:
-- When any input signal changes,
-- compute new result.
------------------------------------------------------------------

add: process(a, b, cin)

variable res_18: vlbit_1d(-1 to 16); -- 18-bit temporary result

constant XOUT: vlbit_1d(0 to 16) := ('X',
'X', 'X', 'X', 'X',
'X', 'X', 'X', 'X',
'X', 'X', 'X', 'X',
'X', 'X', 'X', 'X');

begin

-- Compute the ACCUMULATOR output.

res_18 := add2c (add2c (a, b), '0' & cin);
addout <= res_18(0 to 16);
end process;

register_process: process
begin
wait until clk = '1';
ResReg <= addout(1 to 16);
end process;

------------------------------------------------------------------
-- Concurrent signal assignments:
-- When ACCUMULATOR output or result register changes,
-- schedule new values on processor or memory data out pins.
------------------------------------------------------------------
if GR='1'
pcout <= addout(0) after cout_delay;
pout <= ResReg after reg_delay;
else
mcout <= addout(0) after cout_delay;
mout <= ResReg after reg_delay;

end behavior;
```

```
--------------------------------------------------------------------------
-- Behavioral Model for 8-Bit ADDER.
--
-- Inputs: A leg (8 bits)
-- B leg (8 bits)
-- Carry in (1 bit)
-- Result register clock
--
-- Outputs: Data out (8 bits, latched)
-- Carry out (1 bit, unlatched)
--
--------------------------------------------------------------------------

--------------------------------------------------------------------------
-- Interface declaration:
--------------------------------------------------------------------------

entity add is

-- Generic delays, with default values

 generic (cout_delay: time := 4300 PS; -- Carry out delay
 reg_delay: time := 4200 PS); -- Register delay

-- IO ports:

 port (signal dout: out vlbit_vector(0 to 7);
 signal cout: out vlbit;
 signal a,b: in vlbit_vector(0 to 7);
 signal cin: in vlbit;
 signal clk: in vlbit);

end add;

--------------------------------------------------------------------------
-- Architecture body:
--------------------------------------------------------------------------

architecture behavior of add is

signal addout: vlbit_vector(0 to 8); -- ADD output: cout & dout
signal ResReg: vlbit_vector(0 to 7); -- Result register

 begin
```

```
---------------------------------------------------------------------

-- Concurrent ADD process statement:
-- When any input signal changes,
-- compute new result.
---------------------------------------------------------------------

add: process(a, b, cin)
variable res_10 vlbit_1d(-1 to 8); -- 10-bit temporary result
constant XOUT: vlbit_1d(0 to 8) := ('X','X', 'X', 'X', 'X','X', 'X', 'X','X');
begin

res_18 := add2c (add2c (a, b), '0' & cin);
addout <= res_18(0 to 16);

end process;

---------------------------------------------------------------------
-- Concurrent Register process statement:
-- Load up the result register, but only on rising edge of clock.
---------------------------------------------------------------------

register_process: process
begin
wait until clk = '1';
ResReg <= addout(1 to 16);
end process;

---------------------------------------------------------------------
-- Concurrent signal assignments:
-- When ADDER output or result register changes,
-- schedule new values on data out pins.
---------------------------------------------------------------------

cout <= addout(0) after cout_delay;
dout <= ResReg after reg_delay;
end behavior;
```

```
-----------------------------------------------------------------------
-- Behavioral Model for 6-Bit COUNTER
--
-- Inputs:
-- Clock (1 Bit)
-- Clear (1 Bit)
-- Enable (1 Bit)
--
-- Outputs: Data Out (6 Bits, Unlatched)
-----------------------------------------------------------------------


-----------------------------------------------------------------------
-- Interface Declaration:
-----------------------------------------------------------------------

entity count6b is

 generic(delay:time := 1 ns;
          max:integer:= 20 );

 port (signal enbl : in vlbit;
       signal clk   : in vlbit;
       signal clr: in vlbit;
      signal c : out vlbit_vector(0 to 5):= ('0','0','0','0','0','0'));

 end count6b;
-----------------------------------------------------------------------
 -- COUNTER process statement:
 -----------------------------------------------------------------------

architecture behavior of count6b is

signal temp : vlbit_vector (0 to 31):=('0','0','0','0','0','0',
'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0',
'0','0','0','0','0','0','0','0');
signal zero : vlbit_vector (0 to 31):=('0','0','0','0','0','0','0','0',
'0','0','0','0', '0','0','0','0','0','0','0','0','0','0','0','0', '0','0','0','0',
'0','0', '0','0');
signal unknown : vlbit_vector (0 to 5):=('x','x','x','x','x','x');
signal highimp : vlbit_vector (0 to 5):=('z','z','z','z','z','z');
```

```
-- Compute the COUNTER output.

begin
      process

variable t:integer:=1 ;
variable one:vlbit_vector (0 to 1):=('0','1');
        begin
        wait on clk,enbl;
        if enbl='0' then
        if t=1 then
        t:=0;
        c <= unknown,highimp after delay;
         end if;
-----------------------------------------------------------------------
--counter  counts from up  till 31 when 'clk' becomes high (level sensitive).It resets to 0 once 31 -
--is reached
-----------------------------------------------------------------------
else
        if clk='1' then
        t:=1;
        if clr='1' then
                temp<=addum(zero(1 to 31),one);
                c <= zero(26 to 31) after delay;
        else
        if v1d2int(temp)>max-1 then
                temp<=zero;
        else
                temp<=addum(temp(1 to 31),one);
        end if;

         c <= temp(26 to 31) after delay;
        end if;
end if;

end if;

end process;

end behavior;
```

```
-----------------------------------------------------------------------
-- Behavioral Model for 8-bit LATCH
--
-- Inputs:
-- lat (1 Bit)
-- lin (8-bit input data)
-- Enable (1 Bit)
--
-- Outputs: lout (8 Bits )
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- Interface Declaration:
-----------------------------------------------------------------------


entity latch8b is

 generic(delay:time := 1 ns;
       setup:time := 1 ns);

 port (signal enbl: in vlbit;
     signal lat : in vlbit;
     signal lin  : in vlbit_vector (0 to 7);
     signal lout : out vlbit_vector (0 to 7));


 end latch8b;


-----------------------------------------------------------------------
-- Architecture body:
-----------------------------------------------------------------------

architecture behavior of latch8b is

signal temp : vlbit_vector (0 to 7);
signal highimp : vlbit_vector (0 to 7):=('z','z','z','z','z','z','z','z');
```

```
begin
--------------------------------------------------------------------
 -- LATCH  process statement:
  ---------------------------------------------------------------------
process
variable t,c:integer:=1 ;
 begin

-------------------------------------------------------------------------
--latch output is in high impedance state when enable is deasserted.
--When enbl is asserted, the incoming data is latched in and appears at the output when the
--lat signal is asssserted.
-------------------------------------------------------------------------

 wait on enbl,lat,lin;
  if enbl='0' then
     if t=1 then
      t:=0;
      lout <= highimp after delay;
     end if;
  else
    temp<=lin after setup;
    if lat='0' then
      if c=1 then
        lout <= temp after delay;
        c:=0;
        t:=1;
      end if;
    else
      c:=1;
    end if;
  end if;
 end process;
end behavior;
```

```
-----------------------------------------------------------------------
-- Behavioral Model for 8-bit Two's Complement Multiplier
--
-- Inputs:
-- a  (8-bit input data)
-- b  (8-bit input data)
--
-- Outputs: c  (8 Bits )
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- Interface Declaration:
-----------------------------------------------------------------------


entity mult8bt is

  generic(delay:time := 10 ns);

  port (signal a,b : in vlbit_vector(0 to 7);

        signal c : out vlbit_vector(0 to 7));


  end mult8bt;
-----------------------------------------------------------------------
-- Architecture body:
-----------------------------------------------------------------------

architecture behavior of mult8bt is

signal temp : vlbit_vector (0 to 15);
signal unknown : vlbit_vector (0 to 7):=('x','x','x','x','x','x','x','x');
-
begin
  temp <= mul2c(a,b);
  c <= unknown,temp(0 to 7) after delay;
end behavior;
```

```
-----------------------------------------------------------------------
-- Behavioral Model for 8-bit Two's Complement subtractor
--
-- Inputs:
-- A (8-bit input data)
-- B  (8-bit input data)
-- CLK (1 Bit)
-- Outputs: SOUT (8 Bits )
             COUT (1 bit)
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- Interface Declaration:
-----------------------------------------------------------------------

entity subt is
generic(delay:time:=1ns);
port(A:in vlbit_vector(0 to 15);
B:in vlbit_vector(0 to 15);
CLK:in vlbit;
COUT:out vlbit;
SOUT:out vlbit_vector(0 to 15));
end subt;

-----------------------------------------------------------------------
-- Architecture body:
-----------------------------------------------------------------------
architecture behavior of subt is

signal S:vlbit_vector(0 to 16);
begin

-------------------------------------------------------------------
 -- SUBTRACTOR  process statement:
 -------------------------------------------------------------------
subt:process
begin
-----------------------------------------------------------------------
--subtractor sensitive on clk(positive level sensitive)
-----------------------------------------------------------------------
wait until CLK='1';
S <= sub2c(A,B);
end process;
SOUT <= S(0 to 15) after delay;
COUT <= S(16) after delay;

end behavior;
```

```
----------------------------------------------------------------------
-- Behavioral Model for a 3-to-8 decoder used in the loading
--control unit.

-- Inputs:
-- a (2-bit input data)
--
-- CLK (1 Bit)
-- Outputs: o0 to o8 ( 1 bit each)
----------------------------------------------------------------------
----------------------------------------------------------------------
-- Interface Declaration:
----------------------------------------------------------------------


entity ldec3 is
  generic(delay:time := 1 ns);
  port (signal a : in vlbit_vector(0 to 2);
       signal clk  : in vlbit;
       signal o0   : out vlbit;
       signal o1   : out vlbit;
       signal o2   : out vlbit;
       signal o3   : out vlbit;
       signal o4   : out vlbit;
       signal o5   : out vlbit;
       signal o6   : out vlbit;
       signal o7   : out vlbit);
  end ldec3;
----------------------------------------------------------------------
-- Architecture body:
----------------------------------------------------------------------
architecture behavior of ldec3 is
signal temp : vlbit_vector (0 to 7) := ('0','0','0','0','0','0','0','0');
signal unknown : vlbit_vector (0 to 7):=('x','x','x','x','x','x','x','x');
signal highimp : vlbit_vector (0 to 7):=('z','z','z','z','z','z','z','z');
begin
----------------------------------------------------------------------
 -- DECODER  process statement:
 ---------------------------------------------------------------------


  ldec3_process:process
    variable num:integer;
    begin
----------------------------------------------------------------------
decoder is sensitive on clk (positive level sensitive)
----------------------------------------------------------------------
     temp(0 to 7) <= ('0','0','0','0','0','0','0','0');
```

```
 wait until clk = '1';
    if v1d2int(a)=0 then
temp(0) <='1';
    else
temp(0) <='0';
    end if;
    if v1d2int(a)=1 then
temp(1) <='1';
    else
temp(1) <='0';
    end if;
    if v1d2int(a)=2 then
temp(2) <='1';
    else
temp(2) <='0';
    end if;
    if v1d2int(a)=3 then
temp(3) <='1';
    else
temp(3) <='0';
    end if;
    if v1d2int(a)=4 then
temp(4) <='1';
    else
temp(4) <='0';
    end if;
    if v1d2int(a)=5 then
temp(5) <='1';
    else
temp(5) <='0';
    end if;
    if v1d2int(a)=6 then
temp(6) <='1';
    else
temp(6) <='0';
    end if;
    if v1d2int(a)=7 then
temp(7) <='1';
    else
temp(7) <='0';
    end if;
```

```
    o0 <= temp(0) after delay;
    o1 <= temp(1) after delay;
    o2 <= temp(2) after delay;
    o3 <= temp(3) after delay;
    o4 <= temp(4) after delay;
    o5 <= temp(5) after delay;
    o6 <= temp(6) after delay;
    o7 <= temp(7) after delay;
  end process;
end behavior;
```

Appendix B

Fortran program used to simulate the bilinear transformation algorithm

```
*
* This program estimate the DOA's using the bilinear transformation

      real kr(33,8,8),ki(33,8,8),pm(360)
      real gr(8,8),gi(8,8),wr(8),wi(8),xx1(15),xx2(15)
      real rr(8,8),ri(8,8),ur(8,8),ui(8,8),nx(3,8),yr(8,8),yi(8,8)
      real s1r(15),s1i(15), s2r(15), s2i(15),f(33)
      real nr(3,8), ni(3,8), xr(8),xi(8),zr(64,8),zi(64,8)
      real bb(64), cc(64),br(64,8), bi(64,8),b(8,8)
      real tr(8,8),ti(8,8),ttr(8,8),tti(8,8),gnr(8,8),gni(8,8)
      real fc,li(8,8),lr(8,8)
      integer option
      integer seed, nout, d
      external rnset, rnunf, umach
      seed=345
      pi=acos(-1.)
      print *,'0 for partially coherent, 1 otherwise'
      read *,option
      print *, 'input the number of sensors'
      read *, ns
      print *, 'input the signal power'
      read *, sig1
      print *, 'input the noise power'
      read *, sig2
      stdev1=sqrt(sig1)
      stdev2=sqrt(sig2)
      open(unit=1,file='1.dat',status='new')
      open(unit=2,file='2.dat',status='new')
      open(unit=3,file='3.dat',status='new')
      open(unit=4,file='4.dat',status='new')
      open(unit=5,file='5.dat',status='new')

*
* Initialize the covariance matrices
*
      do 1 i=1,33
      do 1 j=1,ns
      do 1 k=1,ns
      kr(i,j,k)=0.
      ki(i,j,k)=0.
1     continue
      print *,'Covariance Matrix initialized'
*
* Generate signals for sources
*
```

```
seed=123
 call umach(2,nout)
call rnset(seed)
 temp1=rnunf()
seed=seed+2
 phase1=2*pi*temp1
call umach(2,nout)
call rnset(seed)
 temp2=rnunf()
 phase2=2*pi*temp2
 call gauss(stdev1,x,seed)
 xx1(1)=.164*x
 xx2(1)=xx1(1)
 s1r(1)=xx1(1)*cos(phase1)
 s1i(1)=xx1(1)*sin(phase1)
 s2r(1)=s1r(1)
 s2i(1)=s1i(1)
 if (option.eq.0) goto 12
 call gauss(stdev1,x,seed)
 xx2(1)=.164*x
 s2r(1)=xx2(1)*cos(phase2)
 s2i(1)=xx2(1)*sin(phase2)
12 call gauss(stdev1,x,seed)
 xx1(2)=.164*x+1.37*xx1(1)
 xx2(2)=xx1(2)
 s1r(2)=xx1(2)*cos(phase1)
 s1i(2)=xx1(2)*sin(phase1)
 s2r(2)=s1r(2)
 s2i(2)=s1i(2)
 if (option.eq.0) goto 13
 call gauss(stdev1,x,seed)
 xx2(2)=.164*x+1.37*xx2(1)
 s2r(2)=xx2(2)*cos(phase2)
 s2i(2)=xx2(2)*sin(phase2)
13 do 2 i=3,15
 call gauss(stdev1,x,seed)
 xx1(i)=.164*x+1.37*xx1(i-1)-.723*xx1(i-2)
 xx2(i)=xx1(i)
 s1r(i)=10.*xx1(i)*cos(phase1)
 s1i(i)=10.*xx1(i)*sin(phase1)
 s2r(i)=s1r(i)
 s2i(i)=s1i(i)
 if (option.eq.0) goto 2
 call gauss(stdev1,x,seed)
 xx2(i)=.164*x+1.37*xx2(i-1)-.723*xx2(i-2)
 s2r(i)=10.*xx2(i)*cos(phase2)
```

THIS

PAGE

IS

MISSING

IN

ORIGINAL

DOCUMENT

132

```
xx1(k)=xx1(k+1)
xx2(k)=xx2(k+1)
7 continue
call gauss(stdev1,x,seed)
xx1(15)=.164*x+1.37*xx1(14 )-.723*xx1(13 )
xx2(15)=xx1(15)
s1r(15)=xx1(15)*cos(phase1)
s1i(15)=xx1(15)*sin(phase1)
s2r(15)=s1r(15)
s2i(15)=s1i(15)
if (option.eq.0) goto 15
call gauss(stdev1,x,seed)
xx2(15)=.164*x+1.37*xx2(14)-.723*xx2(13)
s2r(15)=xx2(15)*cos(phase2)
s2i(15)=xx2(15)*sin(phase2)
15 do k=1,ns
nx(1,k)=nx(2,k)
nx(2,k)=nx(3,k)
call gauss(stdev2,x,seed)
nx(3,k)=.164*x + 1.37*nx(2,k) - .723*nx(1,k)
nr(3,k)=nx(3,k)*cos(phase)
ni(3,k)=nx(3,k)*sin(phase)
enddo
do 8 k=1,ns
j1=16-k
j2=17-2*k
xr(k)=s1r(j1)+s2r(j2)+nr(3,k)
xi(k)=s1i(j1)+s2i(j2)+ni(3,k)
8 continue
5 continue
*
* Compute the FFT for every sensor output
*
n=64
m=6
do k= 1,ns
do l=1,64
bb(l)=zr(l,k)
cc(l)=zi(l,k)
enddo
call fft (bb,cc,m,n)
do l=1,64
br(l,k)=bb(l)
bi(l,k)=cc(l)
enddo
enddo
```

```
*
* Generate the data covariance matrices
*
 do 9 j=1,33
 do 9 k=1,ns
 do 9 l=1,ns
 kr(j,k,l)=kr(j,k,l)+br(j,k)*br(j,l)+bi(j,k)*bi(j,l)
 ki(j,k,l)=ki(j,k,l)+bi(j,k)*br(j,l)-bi(j,l)*br(j,k)
9 continue
4 continue
 do 10 j=1,33
 do 10 k=1,ns
 do 10 l=1,ns
 kr(j,k,l)=kr(j,k,l)/64.
 ki(j,k,l)=ki(j,k,l)/64.
10 continue
 print *,'Covariance matrix computed'


* Computation of the transformation matrix
*
 b(1,1)=1
 b(1,2)=7
 b(1,3)=21
 b(1,4)=35
 b(1,5)=35
 b(1,6)=21
 b(1,7)=7
 b(1,8)=1
 b(2,1)=1
 b(2,2)=5
 b(2,3)=9
 b(2,4)=5
 b(2,5)=-5
 b(2,6)=-9
 b(2,7)=-5
 b(2,8)=-1
 b(3,1)=1
 b(3,2)=3
 b(3,3)=1
 b(3,4)=-5
 b(3,5)=-5
 b(3,6)=1
 b(3,7)=3
 b(3,8)=1
 b(4,1)=1
```

```
b(4,2)=1
b(4,3)=-3
b(4,4)=-3
b(4,5)=3
b(4,6)=3
b(4,7)=-1
b(4,8)=-1
b(5,1)=1
b(5,2)=-1
b(5,3)=-3
b(5,4)=3
b(5,5)=3
b(5,6)=-3
b(5,7)=-1
b(5,8)=1
b(6,1)=1
b(6,2)=-3
b(6,3)=1
b(6,4)=5
b(6,5)=-5
b(6,6)=-1
b(6,7)=3
b(6,8)=-1
b(7,1)=1
b(7,2)=-5
b(7,3)=9
b(7,4)=-5
b(7,5)=-5
b(7,6)=9
b(7,7)=-5
b(7,8)=1
b(8,1)=1
b(8,2)=-7
b(8,3)=21
b(8,4)=-35
b(8,5)=35
b(8,6)=-21
b(8,7)=7
b(8,8)=-1
print *,'Transformation matrix generated'
*
* Initialize the covariance matrix
*
do i =1,ns
do j =1,ns
 gr(i,j)=0.
```

```
gi(i,j)=0.
gnr(i,j)=0.
gni(i,j)=0.
enddo
enddo
fc=1.
a=2./32.
print *,'Covariance matrices initialized'
print *,a
*
* Computation of G and Gn
*
do i=1,33
print *,float(i)
f(i)=float(i)*a
print *,f(i)
do j=1,ns
do k=1,ns
tr(j,k)=b(j,k)*(2*fc/f(i))**(k-1)*int(cos(float(k-1)*pi/2.))
ti(j,k)=-b(j,k)*(2*fc/f(i))**(k-1)*int(sin(float(k-1)*pi/2.))
enddo
enddo
do j=1,ns
do k=1,ns
ttr(j,k)=0.
tti(j,k)=0.
do l=1,ns
ttr(j,k)=ttr(j,k)+tr(j,l)*kr(i,l,k)-ti(j,l)*ki(i,l,k)
tti(j,k)=tti(j,k)+tr(j,l)*ki(i,l,k)+ti(j,l)*kr(i,l,k)
enddo
enddo
enddo
do j=1,ns
do k=1,ns
do l=1,ns
gr(j,k)=gr(j,k)+ttr(j,l)*tr(k,l)+tti(j,l)*ti(k,l)
gi(j,k)=gi(j,k)-ttr(j,l)*ti(k,l)+tti(j,l)*tr(k,l)
gnr(j,k)=gnr(j,k)+tr(j,l)*tr(k,l)+ti(j,l)*ti(k,l)
gni(j,k)=gni(j,k)-tr(j,l)*ti(k,l)+ti(j,l)*tr(k,l)
enddo
enddo
enddo
enddo
print *,'Printing Resultant matrices'
do i = 1, ns
print *, (gr(i,k),k=1,ns)
```

```
print *, (gi(i,k),k=1,ns)
enddo
print *,'========================================'
do i =1,ns
print *, (gnr(i,k),k=1,ns)
print *, (gni(i,k),k=1,ns)
enddo
call cho (gnr,gni,lr,li,ns)
print *,'Cholesky decomposed'
print *,'Printing Triangular matrix '
do i = 1, ns
print *,(lr(i,k),k=1,ns)
print *, (li(i,k),k=1,ns)
enddo
print *,'========================================'
call tdat (lr,li,gr,gi,yr,yi,ns)
print *,'Printing Eigendecomposed matrix '
do i = 1, ns
print *, (yr(i,k),k=1,ns)
print *, (yi(i,k),k=1,ns)
enddo
print *,'========================================'

call hhc (yr,yi,rr,ri,ur,ui,ns)
 print *,'Printing output of householder transformation '
do i = 1, ns
print *,(rr(i,k),k=1,ns)
print *, (ri(i,k),k=1,ns)
enddo
print *,'========================================'
do i=1,ns
write(3,100) (rr(i,j),j=1,ns)
enddo
write(3,*) ' '
do i=1,ns
write(3,100) (ri(i,j),j=1,ns)
enddo


print *,'2'
 close(unit=1)
close(unit=2)
close(unit=3)
*
 call qrc (rr,ri,tr,ti,ur,ui,ns)
 print *,'Printing output of QR transformation '
```

```
 do i = 1, ns
 print *,(ur(i,k),k=1,ns)
 print *, (ui(i,k),k=1,ns)
 enddo
 print *,'========================================'
do i=1,ns
write(4,100) (tr(i,j),j=1,ns)
enddo
write(4,*) ' '
do i=1,ns
write(4,100) (ti(i,j),j=1,ns)
enddo
print *,'3'
close(unit=4)
100format(2x,8f11.2)

d=2
 call power (ur,ui,pm,d,ns)
do i=1,90
jj=i-1
write(5,*) pm(i)
enddo
write(5,*) ' '
close(unit=5)
 stop
 end




*
* This subroutine performs the Cholesky decomposition
*
 subroutine cho(cnr,cni,lr,li,n)
 real cnr(8,8),cni(8,8),lr(8,8),li(8,8)
 print *,'Starting to decompose Cholesky'
 do i= 1,n
 do j= 1,n
 lr(i,j)=cnr(i,j)
 li(i,j)=cni(i,j)
 enddo
 print *,lr(i,i),li(i,i)
 enddo
 do 1 k=1,n
 do 2 i=1,k-1
 sr=0.
```

```
si=0.
do 3 j=1,i-1
sr=sr+lr(i,j)*lr(k,j)+li(i,j)*li(k,j)
si=si+lr(i,j)*li(k,j)-li(i,j)*lr(k,j)
3 continue
*
print *,lr(i,i),li(i,i)
lr(k,i)=(lr(k,i)-sr)/lr(i,i)
li(k,i)=(li(k,i)-si)/lr(i,i)
2 continue
sr=0.
si=0.
do 4 j=1,k-1
sr=sr+lr(k,j)*lr(k,j)+li(k,j)*li(k,j)
4 continue
t=abs(cnr(k,k)-sr)
if (t.gt.0.) then
lr(k,k)=sqrt(t)
else
lr(k,k)=0.
endif
li(k,k)=0.
1 continue
do i= 1,n
do j= i+1,n
lr(i,j)=0.
li(i,j)=0.
enddo
enddo
return
end

*
* This subroutine performs the eigendecomposition of
* (G,Gn) to (Y,I)
*
subroutine tdat(lr,li,cr,ci,yr,yi,n)
real lr(8,8),li(8,8),cr(8,8),ci(8,8),yr(8,8),yi(8,8)
real xr(8,8),xi(8,8)
do j=1,n
print *,'Lr value is'
print *,lr(j,j)
xr(1,j)= cr(1,j)/lr(1,1)
xi(1,j)= ci(1,j)/lr(1,1)
enddo
do 1 i=2,n
```

```
 do 1 j=1,n
 sr=0.
 si=0.
 do 2 k=1,i-1
 sr=sr+lr(i,k)*xr(k,j)-li(i,k)*xi(k,j)
 si=si+lr(i,k)*xi(k,j)+li(i,k)*xr(k,j)
2 continue
print *,lr(i,i)
 xr(i,j)=(cr(i,j)-sr)/lr(i,i)
 xi(i,j)=(ci(i,j)-si)/lr(i,i)
1 continue
 do j=1,n
 yr(1,j)= xr(j,1)/lr(1,1)
 yi(1,j)= xi(j,1)/lr(1,1)
 enddo
 do 3 i=2,n
 do 3 j=1,n
 sr=0.
 si=0.
 do 4 k=1,i-1
 sr=sr+lr(i,k)*yr(k,j)+li(i,k)*yi(k,j)
 si=si+lr(i,k)*yi(k,j)-li(i,k)*yr(k,j)
4 continue
 yr(i,j)=(xr(j,i)-sr)/lr(i,i)
 yi(i,j)=(xi(j,i)-si)/lr(i,i)
3 continue
 return
 end


* Householders Algorithm for complex data

 subroutine hhc (yr,yi,rr,ri,ur,ui,n)
 real rr(n,n),ri(n,n),ur(n,n),ui(n,n),wr(8),wi(8)
 real yr(n,n),yi(n,n)
*
* Initialisation for the eigenvectors
*
 do 1 i=1,n
 do 2 j=1,n
 ur(i,j)=0.0
 ui(i,j)=0.0
2 continue
1 continue
 do 3 i=1,n
 ur(i,i)=1
```

```
 ui(i,i)=0
3 continue
*
* Compute householder's transformations'
*
 do 4 i=1,n-2
 r1=0.0
 do 5 j=i+1,n
 r1=r1+yr(j,i)*yr(j,i)+yi(j,i)*yi(j,i)
5 continue
 d=sqrt(yr(i+1,i)*yr(i+1,i)+yi(i+1,i)*yi(i+1,i))
 r1=sqrt(r1)/d
 wr(i)=yr(i+1,i)+r1*yr(i+1,i)
 wi(i)=yi(i+1,i)+r1*yi(i+1,i)
 yr(i+1,i)=-r1*yr(i+1,i)
 yi(i+1,i)=-r1*yi(i+1,i)
 yr(i,i+1)=yr(i+1,i)
 yi(i,i+1)=-yi(i+1,i)
 do 6 j=i+1,n-1
 wr(j)=yr(j+1,i)
 wi(j)=yi(j+1,i)
6 continue
 c=0.
 do 18 j=i,n-1
 c=c+wr(j)*wr(j)+wi(j)*wi(j)
18 continue
 c=c/2
*
* Compute the update covariance data matrix for every
* transformation
*
 do 7 j=i+2,n
 yr(i,j)=0.0
 yr(j,i)=0.0
 yi(i,j)=0.0
 yi(j,i)=0.0
7 continue
 do 8 j=i+1,n
 d1=0.0
 d2=0.0
 do 9 k=i+1,n
 d1=d1+wr(k-1)*yr(k,j)+wi(k-1)*yi(k,j)
 d2=d2+wr(k-1)*yi(k,j)-wi(k-1)*yr(k,j)
9 continue
 d1=d1/c
 d2=d2/c
```

```
do 10 k=i+1,n
yr(k,j)=yr(k,j)-(wr(k-1)*d1-wi(k-1)*d2)
yi(k,j)=yi(k,j)-(wr(k-1)*d2+wi(k-1)*d1)
10 continue
8 continue
do 11 j=i+1,n
d1=0.0
d2=0.0
do 12 k=i+1,n
d1=d1+wr(k-1)*yr(j,k)-wi(k-1)*yi(j,k)
d2=d2+wr(k-1)*yi(j,k)+wi(k-1)*yr(j,k)
12 continue
d1=d1/c
d2=d2/c
do 13 k=i+1,n
yr(j,k)=yr(j,k)-(d1*wr(k-1)+d2*wi(k-1))
yi(j,k)=yi(j,k)-(d2*wr(k-1)-d1*wi(k-1))
13 continue
11 continue
*
* Compute the eigenvectors
*
do 14 j=i,n
d1=0.0
d2=0.0
do 15 k=i+1,n
d1=d1+wr(k-1)*ur(k,j)+wi(k-1)*ui(k,j)
d2=d2+wr(k-1)*ui(k,j)-wi(k-1)*ur(k,j)
15 continue
d1=d1/c
d2=d2/c
do 16 k=i+1,n
ur(k,j)=ur(k,j)-(wr(k-1)*d1-wi(k-1)*d2)
ui(k,j)=ui(k,j)-(wr(k-1)*d2+wi(k-1)*d1)
16 continue
14 continue
4 continue
*
*
do 17 i=1,n
do 17 j=1,n
rr(i,j)=yr(i,j)
ri(i,j)=yi(i,j)
17 continue
return
end
```

* This subroutine computes the QR transformation of the data

```
subroutine qrc(rr,ri,tr,ti,ur,ui,n)
real tr(n,n),ti(n,n),qr(8,8), qi(8,8)
real rr(n,n),ri(n,n),ur(n,n),ui(n,n)
do 1 i=1,n
do 1 j=1,n
tr(i,j)=rr(i,j)
ti(i,j)=ri(i,j)
1 continue
iter=0
15 iter=iter+1
do 2 i=1,n
do 2 j=1,n
qr(i,j)=0
qi(i,j)=0
2 continue
do 3 i=1,n
qr(i,i)=1
qi(i,i)=0.
3 continue
y=tr(1,1)
do 4 i=1,n-1
x=tr(i+1,i)*tr(i+1,i)+ti(i+1,i)*ti(i+1,i)
if (x.eq.0.) then
y=tr(i+1,i+1)
else
x=x+y*y
x=sqrt(x)
pr11=y/x
pi11=0.
pr12=tr(i+1,i)/x
pi12=-ti(i+1,i)/x
pr21=-pr12
pi21=pi12
pr22=pr11
pi22=0.
do 7 j=1,n
cr1=pr11*ur(i,j)-pi11*ui(i,j)+pr12*ur(i+1,j)-pi12*ui(i+1,j)
ci1=pr11*ui(i,j)+pi11*ur(i,j)+pr12*ui(i+1,j)+pi12*ur(i+1,j)
cr2=pr21*ur(i,j)-pi21*ui(i,j)+pr22*ur(i+1,j)-pi22*ui(i+1,j)
ci2=pr21*ui(i,j)+pi21*ur(i,j)+pr22*ui(i+1,j)+pi22*ur(i+1,j)
ur(i,j)=cr1
ui(i,j)=ci1
```

```
ur(i+1,j)=cr2
ui(i+1,j)=ci2
7 continue
do 8 j=1,n
cr1=pr11*qr(i,j)-pi11*qi(i,j)+pr12*qr(i+1,j)-pi12*qi(i+1,j)
ci1=pr11*qi(i,j)+pi11*qr(i,j)+pr12*qi(i+1,j)+pi12*qr(i+1,j)
cr2=pr21*qr(i,j)-pi21*qi(i,j)+pr22*qr(i+1,j)-pi22*qi(i+1,j)
ci2=pr21*qi(i,j)+pi21*qr(i,j)+pr22*qi(i+1,j)+pi22*qr(i+1,j)
qr(i,j)=cr1
qi(i,j)=ci1
qr(i+1,j)=cr2
qi(i+1,j)=ci2
8 continue
j=i+1
y=pr21*tr(i,j)-pi21*ti(i,j)+pr22*tr(i+1,j)-pi22*ti(i+1,j)
endif
4 continue
do 9 i=1,n
do 9 j=1,n
rr(i,j)=0
ri(i,j)=0
9 continue
do 10 i=1,n
do 10 j=1,n
do 10 k=1,n
rr(i,j)=rr(i,j)+qr(i,k)*tr(k,j)-qi(i,k)*ti(k,j)
ri(i,j)=ri(i,j)+qi(i,k)*tr(k,j)+qr(i,k)*ti(k,j)
10 continue
do 11 i=1,n
do 11 j=1,n
tr(i,j)=0
ti(i,j)=0
11 continue
do 12 i=1,n
do 12 j=1,n
do 12 k=1,n
tr(i,j)=tr(i,j)+rr(i,k)*qr(j,k)+ri(i,k)*qi(j,k)
ti(i,j)=ti(i,j)+ri(i,k)*qr(j,k)-rr(i,k)*qi(j,k)
12 continue
s=0.
do 13 i=1,n-1
j=i+1
s=s+tr(j,i)*tr(j,i)+ti(j,i)*ti(j,i)
13 continue

print *, 'QR matrix'
```

```
do i=1,n
print 100 , (qr(i,j),j=1,n)
print 100 , (qi(i,j),j=1,n)
enddo

print *, 'U matrix'
do i=1,n
print 100 , (ur(i,j),j=1,n)
print 100 , (ui(i,j),j=1,n)
enddo


do j=3,n
do i=1,2
sr=0.
si=0.
do k=1,n
sr=sr+ur(j,k)*ur(i,k)
sr=sr+ui(j,k)*ui(i,k)
si=si+ur(j,k)*ui(i,k)
si=si-ui(j,k)*ur(i,k)
enddo
enddo
print *,j,sr,si
enddo
100 format(2x,8f11.2)
print *,'QR iteration No: ',iter
if (iter.le.20) goto 15
return
end



*
* This subroutine estimates the DOA's using MUSIC'
* starts li 767

subroutine power(ur,ui,pm,d,n)
integer d
real ur(8,8),ui(8,8),pm(91)
real sr,se,si
a=0.
b=0.
d=2
print *, 'D value is', d
do k=1,n
```

```
a=a+ur(1,k)*ur(1,k)+ui(1,k)*ui(1,k)
enddo
a=sqrt(a)
a=0.
b=0
do k=1,n
a=a+ur(1,k)*ur(2,k)+ui(1,k)*ui(2,k)
b=b+ur(1,k)*ui(2,k)-ui(1,k)*ur(2,k)
enddo
do k= 1,n
ur(2,k)=ur(2,k)-(a*ur(1,k)-b*ui(1,k))
ui(2,k)=ui(2,k)-(a*ui(1,k)+b*ur(1,k))
enddo
a=0.
do k=1,n
a=a+ur(2,k)*ur(2,k)+ui(2,k)*ui(2,k)
enddo
a=sqrt(a)

do j=3,n
a1=0
b1=0
a2=0
b2=0
do k=1,n
a1=a1+ur(1,k)*ur(j,k)+ui(1,k)*ui(j,k)
b1=b1+ur(1,k)*ui(j,k)-ui(1,k)*ur(j,k)
a2=a2+ur(2,k)*ur(j,k)+ui(2,k)*ui(j,k)
b2=b2+ur(2,k)*ui(j,k)-ui(2,k)*ur(j,k)
enddo
do k=1,n
ur(j,k)=ur(j,k)-(a1*ur(1,k)-b1*ui(1,k))-(a2*ur(2,k)-b2*ui(2,k))
ui(j,k)=ui(j,k)-(a1*ui(1,k)+b1*ur(1,k))-(a2*ui(2,k)+b2*ur(2,k))
enddo
enddo

do j=3,n
do i=1,2
sr=0.
si=0.
do k=1,n
sr=sr+ur(j,k)*ur(i,k)
sr=sr+ui(j,k)*ui(i,k)
si=si+ur(j,k)*ui(i,k)
si=si-ui(j,k)*ur(i,k)
enddo
```

```
 enddo
 print *,j,sr,si
print *, 'Finished inner loop of first part'
 enddo

 pi=acos(-1.)
 do 1 i=2,90
 theta=((float(i)-1)/180.)* pi
 pm(i)=0.
 print *, d ,n
 do 2 j=3,n
 sr=0.
 si=0.
se=0.
 do 3 k=1,n
do 4 l=1,k-1
se=pi*sin(theta)
4continue
 sr=sr+ur(j,k)*se
 si=si+ui(j,k)*se
3 continue
 pm(i)=pm(i)+sr*sr+si*si
2 continue
 jj=i-1
print *, ' PM(i) value is: ',pm(i)
 pm(i)=10.*ALOG(1./pm(i))/ALOG(10.)
 print *,jj, pm(i)
1 continue
 return
 end


 subroutine fft(A,D,M,N)
 dimension A(N),D(N)
 NV2=N/2
 NM1=N-1
 J=1
 do 7 I=1,NM1
 if(I.ge.J) goto 5
 T=A(J)
 Z=D(J)
 A(J)=A(I)
 D(J)=D(I)
 A(I)=T
 D(I)=Z
 5 K=NV2
```

```
6 if(K.ge.J) goto 7
J=J-K
K=K/2
goto 6
7 J=J+K
PI=3.141592653589793
do 20 L=1,M
LE=2**L
LE1=LE/2
U1=1.
U2=0.
W1=COS(PI/LE1)
W2=-SIN(PI/LE1)
do 20 J=1,LE1
do 10 I=J,N,LE
IP=I+LE1
T1=A(IP)*U1-D(IP)*U2
T2=A(IP)*U2+D(IP)*U1
A(IP)=A(I)-T1
D(IP)=D(I)-T2
A(I)=A(I)+T1
10 D(I)=D(I)+T2
U1=U1*W1-U2*W2
20 U2=U1*W2+U2*W1
return
end
```

```
subroutine gauss(sdev,a,AI)
real sdev,a
integer AI
call umach(2,nout)
call rnset(iseed)
s=0.
do 1 ii=1,12
s=s+rnunf()
1continue
iseed=iseed+1
a=(s-6.)*sdev
call umach(2,nout)
call rnset(iseed)
return
end
```

Bibliography

[1] J. H. Wilkinson, "The algebriac eigenvalue problem," Clarendon Press, Oxford, Chapter 4, 1965

[2] J. Capon, "High resolution frequency-wavenumber spectrum analysis. Proc. IEEE, Vol. 57, pp. 1408-1418, Aug. 1969.

[3] J. P. Burg, "Maximum entropy spectral analysis," Proc. 37th Annual International SEG Meeting Oklahoma City, OK, 1967

[4] R. O. Schmith, "Multiple emitter location and signal parameter estimation," IEEE Trans. on Antennas and Propagation, Vol AP-34 No. 3., pp. 276-280, Mar. 1986.

[5] R. Roy and T. Kailath, "ESPRIT-Estimation of signal parameters via rotational invariance techniques", IEEE Trans. Acoustic, Speech and Signal Processing, Vol. 37, No. 7, pp. 984-995, July 1989.

[6] C. H. Knapp and G. C. Carter, "The generalized correlation...", IEEE Trans. ASSP, VOL. 24, No. 4, pp. 320-237, 1976.

[7] W. J. Bangs and P. Schultheiss, "Space-Time processing...", in *Signal Processing,* J. W. R. Griffiths et al, Eds. New York, Academic Press, pp. 577-590, 1973.

[8] W. R. Hahn and S. A. Tretter, "Optimum processing for ...", IEEE Trans. IT, VOL. 19, No. 5, pp. 608-614.

[9] M. Wax and T. Kailath, "Optimum localizations of multiple source by passive arrays", in proc. IEEE Trans. Acoustic, Speech and Signal Processing vol. ASSP-31, No. 5, pp. 1210-1218, Oct. 1983.

[10] B. Porat and B. Frienlander, "Estimation of spatial and spectral parameters of multiple sources", IEEE Trans. on Information Theory, vol. IT-29, pp. 412-425, May 1983.

[11] A. Nehorai, G. Su, M. Morf, "Estimation of time difference of arrivals for multiple ARMA sources by pole decomposition", IEEE Trans. Acoustic, Speech and Signal Processing, vol. ASSP-31, pp. 1478-1491, Dec. 1983.

[12] M. Wax T. J. Shan and T.Kailath, "Spatio-temporal spectral analysis by eigenstructure method", IEEE Trans. Acoustic, Speech and Signal Processing, vol. ASSP-32, No. 4, Aug. 1984.

[13] H. Wang and M. Kaveh, "Estimation of angles-of -arrival for wide-band sources", IEEE Trans. Acoustic, Speech and Signal Processing, pp. 7.5.1-7.5.4, Mar. 19-21, 1984.

[14] H. Wang and M. Kaveh, "Coherent Signal Subspace processing for the detection and estimation of angle of arrival of multiple wide-band sources", IEEE Trans. Acoustic, Speech and Signal Processing, vol. ASSP-33, No. 4, pp. 823-831, Aug. 1985.

[15] Arnab K. Shaw and Ramdas Kumaresan, "Estimation of angles of arrivals of broadband signals", IEEE ICASSP-87, pp.2296-2299, 1987.

[16] *DSP 56000 Simulator Reference Manual* Motorola Inc. 1992

[17] D. Spielman, A. Paulraj, "Performance analysis of the MUSIC algorithm," in proc. IEEE Conference Acoustic, Speech and Signal Processing, Tokyo, Japan, pp 1909-1912, Apr 1986

[18] Kevin M. Buckley and LLoyd J. Griffiths, "Broad-band signal-subspace spatial-spectrum (BASE-ALE) estimation", IEEE Trans. on Acoustics, Speech, and Signal Processing, VOL. 36, No. 7, July 1988.

[19] *Powerview Viewdraw reference manual* Viewlogic Inc. 1991

[20] *Lxcells users guide - Software Version 5.3_1* Mentor Graphics Corp. 1992

[21] *Led users guide - Software Version 5.3_1* Mentor Graphics Corp. 1992

[22] *Lsim users guide - Software Version 5.3_1* Mentor Graphics Corp. 1992

[23]*AutoCells users guide-Software Version 5.3_1* Mentor Graphics Corp. 1992

[24] Ma, G.K, and Taylor F.J., "Multiplier policies for Digital Signal Processing" IEEE ASSP Magazine. January 1990

[25] Chua O. H. and Eldin A.G. "Synthesis algorithms for multipliers used in ASIC design" NASA symposium on VLSI Design 1993

[26] M. Mano. "Computer System Architecture", Prentice Hall 1988.